FACHHOCHSCHULE KONSTANZ
FACHBEREICH INFORMATIK

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (FH)

an der

Fachhochschule Konstanz
Hochschule für Technik, Wirtschaft und Gestaltung

Fachbereich
Informatik

Studiengang
Technische Informatik

Thema: **Entwicklung eines plattformunabhängigen Facility Management Systems unter Verwendung der JAVA Database Connectivity (JDBC)**

Diplomand: **Marc Heller, Richentalstrasse 5, 78462 Konstanz**

Betreuer: Professor Dr. Dipl.-Ing. Reinhard Nürnberg

Dipl.-Ing. Alexander Schaller

Eingereicht: Konstanz, 18. September 2002

Abstract

Die vorliegende Arbeit beschreibt die Entwicklung und Implementierung eines plattformunabhängigen Facility Management Systems[1] für das Captive Office[2] der Firma Nortel Networks Germany. Dieses System stellt ein Individualprojekt dar und wurde spezifisch für Switches[3] vom Typ DMS-100F (Digital Multiplex System) entwickelt, um Mitarbeitern des Captive Office Teams die Möglichkeit zu bieten, über eine Applikation auf der Client-Seite die Konfigurationsdaten des jeweiligen Switches in einer zentralen Datenbank zu sichern, zu modifizieren und auszudrucken.

Bis zu diesem Zeitpunkt existierte weder ein Tool, das diese Aufgabe erfüllte, noch jegliche andere Art der Dokumentation.

[1] Ein Facility Management System dient der rechnergestützten Dokumentation und Verwaltung von Netzwerken wie Telekommunikations-, Daten- und Überwachungsnetzen, sowie deren Geräten.

[2] Testlabor der Firma Nortel Networks Germany.

[3] In der Telekommunikation gebräuchlicher Begriff für eine Vermittlungsstelle.

Inhaltsverzeichnis

Vorwort

Bereits im Rahmen meiner Studienzeit begleitenden Tätigkeit bei der Firma Nortel Networks Germany kam die Problematik der unzureichenden Administration von Konfigurationsdaten der Switches vom Typ DMS-100F (Digital Multiplex System) auf. Die Idee meines Senior Managers Ralf Wiesen, ein plattformunabhängiges System zu entwickeln, das diese Aufgabe erfüllt, hat mich sehr fasziniert und war schliesslich der Anstoss dazu, meine Diplomarbeit in diesem Bereich anzufertigen. Ich war deshalb sehr froh, dass ich die Möglichkeit hatte, unter Anleitung von Herrn Prof. Dr. Dipl.-Ing. Reinhard Nürnberg diese Diplomarbeit zu erstellen.
In diesem Sinne möchte ich allen danken, die mich während der Entstehung dieser Arbeit unterstützt haben.

Ich danke Herrn Prof. Dr. Dipl.-Ing. Reinhard Nürnberg und Herrn Dipl.-Ing. Alexander Schaller, die meine Arbeit stets mit konstruktiver Kritik und inhaltlichen Anregungen begleiteten. Beide hatten stets ein offenes Ohr für meine Belange und lieferten durch Vorschläge und Anregungen die Grundlage meiner Arbeit.
Des Weiteren danke ich Herrn Dipl.-Phys. Ralf Brunberg für die Korrekturlesung meiner Arbeit sowie meinem Kommilitonen Ralf Steppacher, der zum selben Zeitpunkt seine Diplomarbeit angefertigt hat. Die Diskussionen mit ihnen waren stets sehr hilfreich, um meine Ideen weiterzuentwickeln und neue Anstösse zu erhalten. Ebenfalls möchte ich mich bei Frau Nicole Reiser und Herrn Jerôme Ernsberger für deren Ratschläge in Bezug auf das Layout dieser Arbeit bedanken.
Ganz besonderer Dank gebührt auch meiner Freundin Carmen Sütterlin und meiner Familie für die moralische Unterstützung, besonders in Phasen des schleppenden Fortgangs meiner Arbeit. Dabei möchte ich nicht versäumen, meinen Eltern herzlichen Dank dafür auszusprechen, dass sie so manches Opfer gebracht haben, um mir die Ausbildung an der Fachhochschule Konstanz überhaupt zu ermöglichen. Ohne ihren Rückhalt wäre es ungleich schwieriger gewesen, das Projekt „Studium“ zu einem erfolgreichen Abschluss zu bringen.

Abkürzungsverzeichnis

API	Abstract Programming Interface
AWT	Abstract Window Toolkit
CAD	Computer Aided Design
CAFM	Computer Aided Facility Management
CANFM	Computer Aided Network Facility Management
CLI	Call Level Interface
COCMS	Captive Office Configuration Management System
DBMS	Database Management System
DCL	Data Control Language
DDL	Data Definition Language
DLL	Dynamic Link Library
DML	Data Manipulation Language
DMS	Digital Multiplex System
EDV	Elektronische Datenverarbeitung
GUI	Graphical User Interface
IP	Internet Protocol
JDBC	Java Database Connectivity
JDK	Java Development Kit
JFC	Java Foundation Classes
JVM	Java Virtual Machine
MD	Manufacturing Discontinued
MVC	Model-View-Controller
ODBC	Open Database Connectivity
OS	Operating System
PEC	Product Engineering Code
PLAF	Pluggable Look and Feel
RMI	Remote Method Invocation
SMA	Separable Model Architecture
SQL	Standard Query Language
TCP	Transmission Control Protocol
UI	User Interface
UML	Unified Modeling Language

Abbildungsverzeichnis

Abbildungen:

Tabellen:

1 Einleitung

Im Bereich der computerunterstützten Planung und Verwaltung von Gebäuden, Flächen und technischen Ausrüstungen haben sich seit Ende der 80er Jahre die sogenannten Computer Aided Facility Management Systeme (CAFM) etabliert.
In Computer Aided Facility Management Systemen werden sowohl die Orte (Standorte, Einbauorte) als auch die spezifischen Eigenschaften (z.B. technische, funktionale, organisatorische, usw.) von Objekten, sogenannten Facilities, abgebildet.
CAFM-Systeme stellen oft bezüglich ihrer Softwarearchitektur eine Kombination von CAD-Systemen (Computer Aided Design) und Datenbankmanagementsystemen (DBMS) dar. Facilities werden hierbei in ihrem geographischen, bzw. geometrischen Kontext auf der Basis von digitalen Karten oder Zeichnungen visualisiert und binden an eine oder mehrere Datenbanken an, welche die speziellen Eigenschaften und Parameter der Facilities verwalten.
Einen den Anwendungsbereich „Netzwerke“ betreffenden Spezialfall von CAFM-Systemen bilden die Computer Aided Network Facility Management Systeme (CANFM). Diese dienen zur rechnergestützten Planung, Dokumentation und Verwaltung von Netzwerken wie Telekommunikations-, Daten- und Überwachungsnetzen. Bei dieser Form von Systemen werden alle aktiven und passiven Netzwerkkomponenten[1] und deren Verbindungen untereinander im Kontext der Netzumgebung abgebildet.
Eine spezialisierte Form dieser CANFM-Systeme steht mit dem in dieser Diplomarbeit entwickelten Captive Office Configuration Management System (COCMS) zur Verfügung.

[1] Aktive Netzwerkkomponenten: Hub, Switch, Router, usw.
Passive Netzwerkkomponenten: Anschlussdosen, Patchfelder.

1.1 Gliederung

Im ersten Kapitel dieser Arbeit erfolgt eine kurze Einführung in den generellen Aufbau eines Switches vom Typ DMS-100F und dessen Beschaffenheit.
Darauf aufbauend wird im zweiten Kapitel eine Analyse der Anforderungen an das Facility Management System widergespiegelt, die in Absprache mit Vertretern des Captive Office Teams erstellt wurde. Hierzu zählen Anforderungen an die Benutzeroberfläche, das System, dessen Funktionalität und Handhabung.
Das dritte Kapitel beinhaltet die Grundlagen der in der Implementierung verwendeten Techniken. Insbesondere sind hier die Verwendung der *JAVA Foundation Classes (JFC)* für die Benutzeroberflächengestaltung, die Anbindung der Applikation an die bereits existierende *PostgreSQL*-Datenbank via *JAVA Database Connectivity (JDBC)*, die Datenmodellierung sowie der Einsatz von *Design Patterns* und *JAVA Beans* zu nennen. Kapitel vier und fünf bauen auf den im vorigen Kapitel genannten Grundlagen auf und schildern die exakte Vorgehensweise bei der Entwicklung und Implementierung des Systems. Hierbei werden das System als Ganzes sowie die einzelnen Klassen mit ihrer Funktionalität ausführlich erklärt und anhand diverser Diagramme die Interaktion zwischen diesen verdeutlicht.

1.2 Problemstellung und Ziel der Arbeit

Das Ziel dieser Diplomarbeit besteht darin, ein CANFM-System zu entwickeln, das speziell zur Administration von Konfigurationsdaten der Switches vom Typ DMS-100F dient, wie es im Bereich der Telekommunikation üblich ist. Mithilfe dieses Systems soll es möglich sein, Facilities zu erstellen, zu löschen, und diese, in Abhängigkeit von ihrem Kontext, entsprechend in eine auf einem Webserver existierende Datenbank einzubinden. Des Weiteren sollen dem Benutzer Möglichkeiten der Modifikation von Attributwerten zur Verfügung gestellt werden, durch welche das Objekt und seine Eigenschaften beschrieben werden können.

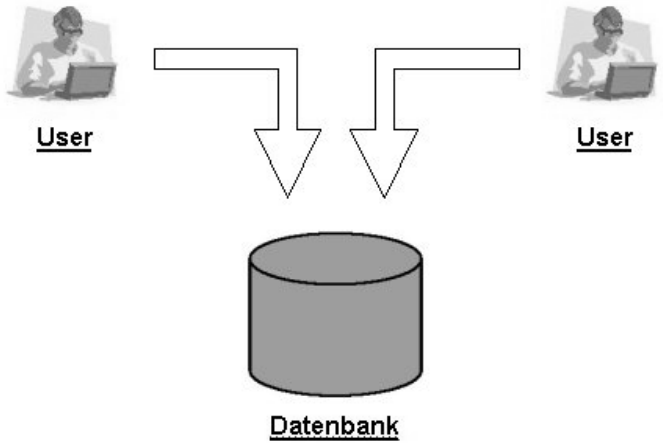


Abbildung 1: Zugriff auf die Datenbank

Um auf den Aufbau und die exakten Aufgaben eines solchen Systems näher eingehen zu können, ist es erforderlich, die generelle Beschaffenheit der Hardware in kurzer Form zu erläutern.

1.3 Aufbau der zu administrierenden Hardware

Ein Switch vom Typ DMS-100F setzt sich in der Regel aus mehreren Schaltschränken (Cabinets) zusammen, welche wiederum in Regale (Shelves) untergliedert sind, die die einzelnen Steckkarten (Cards) beinhalten. Die folgende Abbildung veranschaulicht dies:

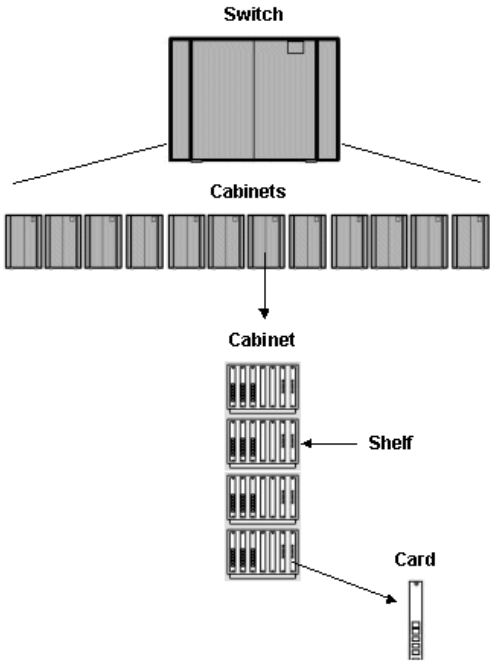


Abbildung 2: Aufbau eines Switches vom Typ DMS-100F

Jedes Cabinet kann von einem anderen Typ sein und somit eine unterschiedliche Anzahl von Shelves beinhalten. Analog hierzu verhält sich die Anzahl der Karten im jeweiligen Shelf, die je nach Typ variieren kann. Aus der vorhergehenden Abbildung lässt sich leicht erkennen, dass bezüglich der Hardware eine hierarchische Gliederung existiert, die vom Switch ausgehend über das Cabinet und Shelf zur Card herunterbricht. Aufgrund dieser Tatsache liegt es nahe, für die Administration von Konfigurationsdaten ein CANFM-System einzusetzen, das in ähnlicher Form strukturiert und in verschiedene Ebenen untergliedert ist.

Bei der Verwendung eines solchen Systems ist es empfehlenswert, das Design der Datenbank ebenfalls an diese Struktur anzulehnen und die Datenbank entsprechend zu modellieren.

2 Analyse des COCMS

2.1 Anforderungen an die Funktionalität

Wie bereits im vorigen Kapitel erwähnt, empfiehlt es sich, das System in verschiedene Ebenen zu untergliedern und für die jeweiligen Facilities
(Switch, Cabinet, Shelf, Card) gesonderte Oberflächen bereitzustellen. Beginnend mit der Ausführung der Applikation auf dem Client befindet sich der Benutzer auf höchster Ebene und erhält somit alle verfügbaren Switches mit deren zugehörigen Attributen in visueller Form aufbereitet. Hierzu müssen alle verfügbaren Daten der Switches bereits bei Programmstart aus der Datenbank ausgelesen werden. Die Switches werden hierbei in einer Liste mit ihren Namen aufgeführt. Die zugehörigen Attributwerte erscheinen mit Auswahl der Listenelemente in einem gesonderten Panel innerhalb der Applikation.

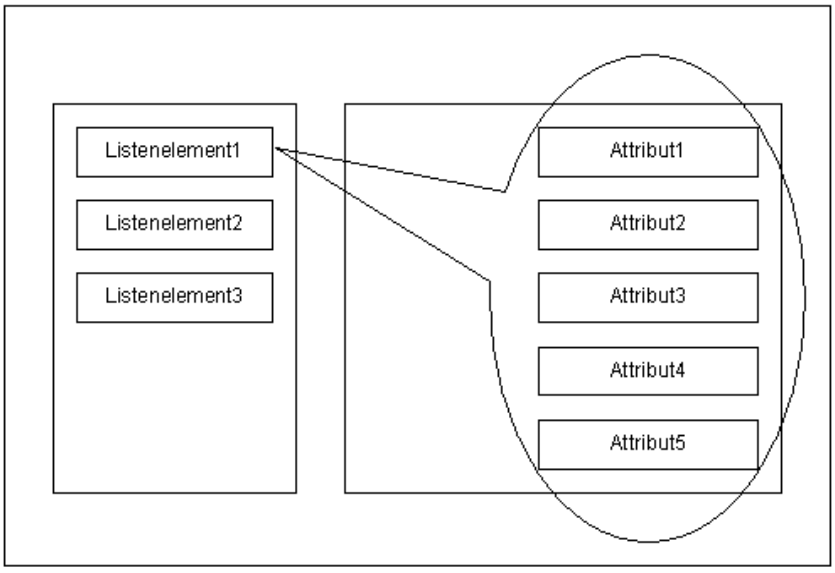


Abbildung 3: Liste mit Elementen (Facilities) und Attributen

Des Weiteren muss nun die Möglichkeit bestehen, dieser Liste Elemente (Facilities) hinzuzufügen bzw. zu entfernen. Hierfür müssen explizit Funktionen bereitgestellt werden, die dies ermöglichen. Zu berücksichtigen ist ebenfalls, dass mit dem Entfernen eines Listenelements alle hiervon abhängigen Elemente auf tieferen Ebenen entfernt werden müssen. Wird z.B. ein Switch der Liste gelöscht, so müssen ebenfalls alle diesem Switch zugeordneten Cabinets, Shelves und Cards entfernt werden. Erfolgt ein doppelter Click mit der Maustaste auf ein Listenelement, so muss die Applikation dafür Sorge tragen, dass in die nächst tiefere Ebene gewechselt wird und die mit dem Listenelement verbundenen Facilities auf der tieferen Ebene dargestellt werden. Erfolgt z.B. ein doppelter Click auf einen Switch in der Liste, so wird in die nächst tiefere Ebene gewechselt und alle dem Switch zugeordneten Cabinets mit deren Attributen präsentiert. Ebenso muss die Möglichkeit bestehen, aus dieser Ebene heraus in die wiederum höhere Ebene zu wechseln. Auch hierfür muss eine Funktion implementiert werden. Die Realisierung der Ebenen für Shelves und Cards muss hierzu analog erfolgen. Alle Attributwerte der jeweiligen Facilities müssen die Möglichkeit bieten, modifizierbar zu sein. Wird eine Änderung eines Attributwertes vorgenommen, so liegt diese Änderung zu Beginn nur lokal auf der jeweiligen Maschine vor. Erst mit dem expliziten Auslösen einer Transaktion werden die Änderungen in die Datenbank übernommen. Hingegen ist es auch erforderlich, Änderungen, die in lokaler Form vorgenommen werden, wieder rückgängig machen zu können. Auch hierfür muss eine Funktion bereitgestellt werden. Weiterhin empfiehlt es sich, den Pfad der Traversierung, der Aufschluss über die aktuelle Position innerhalb der baumförmigen Struktur der Datenbank gibt, in Form einer Zeichenkette auszugeben. Nur so lässt sich eine übersichtliche Navigation innerhalb der Facilities gewährleisten. Abschliessend muss die Möglichkeit bestehen, die Facilities mitsamt ihren Attributen zu drucken. Je nach selektiertem Listenelement soll dieses mit seinen Attributen gedruckt werden.

2.2 Anforderungen an die Benutzeroberfläche

Dem Benutzer soll eine möglichst einfache Art der Administration von Konfigurationsdaten dargeboten werden, d.h. dieser soll nicht mit Informationen überhäuft werden. Hierfür ist es sinnvoll, eine der Realität angelehnte hierarchische Strukturierung des Systems und der Benutzeroberfläche vorzunehmen, die jeweils nur die gewünschten Daten auf der jeweiligen Ebene präsentiert. Die Benutzeroberfläche soll zudem die Möglichkeit bieten, während der Laufzeit in ihrem Aussehen (*Look & Feel*) sowie in ihrer Sprache (Zeichensatz) variiert zu werden.

2.3 Anforderungen an das System

Um die Applikation einem möglichst breiten Anwenderkreis zugänglich zu machen, soll diese plattformunabhängig entwickelt werden und möglichst so erweiterbar sein, dass der Aufruf über einen Browser von einem beliebigen Arbeitsplatz aus erfolgen kann. Zudem muss bei der simultanen Benutzung des Systems durch mehrere Anwender eine Inkonsistenz beim Zugriff auf die zentrale Datenbank ausgeschlossen werden. Besonders wichtig ist die modulare Entwicklung des Systems, so dass dieses jederzeit in Form zusätzlicher Module weiterentwickelt werden kann.

2.4 Produktvoraussetzungen

2.4.1 Hardware

- PC oder UNIX-Workstation.
- Rechnerleistung entsprechend 233 MHz oder höher mit mindestens 128 MB Arbeitsspeicher.

2.4.2 Software

- Betriebssystem mit grafischer Arbeitsumgebung.
- *JAVA Virtual Machine (JVM).*

Das nachstehende Use-Case-Diagramm fasst das System in seiner Funktionalität zusammen und zeigt die in der Analysephase definierten Anforderderungen:

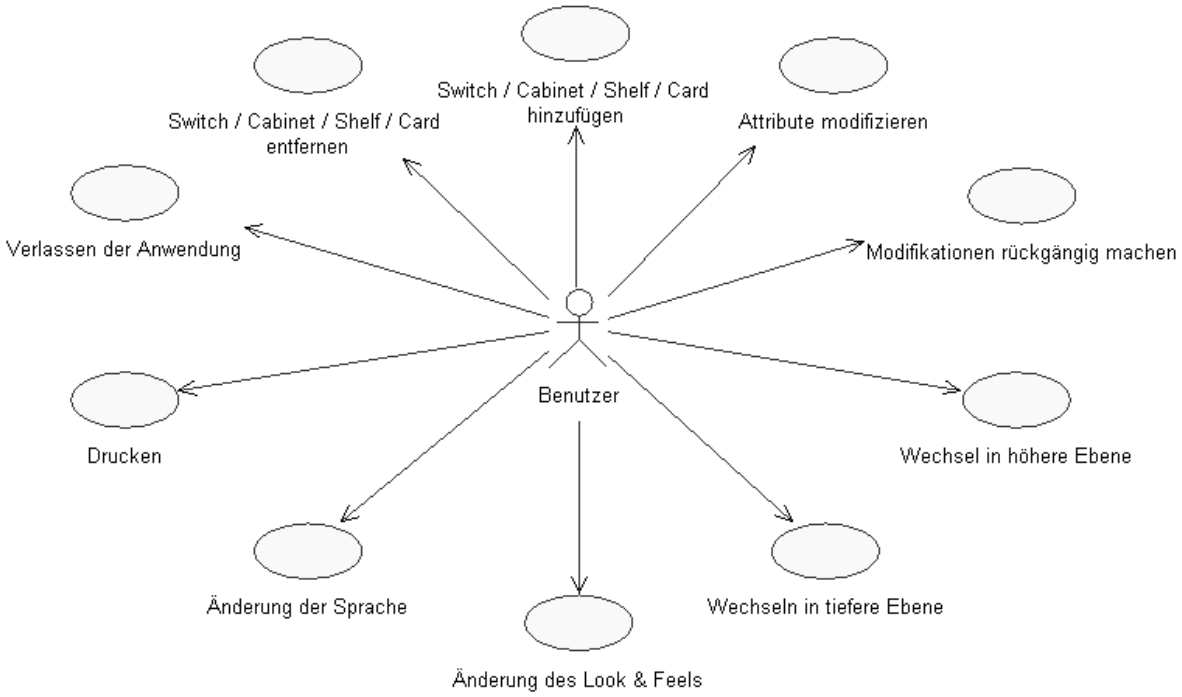


Abbildung 4: Use-Case-Diagramm des Systems

3 Technische Grundlagen

Die in diesem Kapitel beschriebenen Themen stellen die Grundlage der in der Implementierung verwendeten Techniken dar. An erster Stelle wird ein Überblick über die Systemarchitektur des Captive Office Configuration Management Systems gegeben. Anschliessend wird die Verwendung der *JAVA Foundation Classes*, insbesondere *JAVA Swing* für die Gestaltung der Benutzeroberfläche, der *JAVA Database Connectivity* für die Anbindung der Applikation an die bereits vorhandene *PostgreSQL*-Datenbank sowie der Einsatz von *Design Patterns*, insbesondere das *Observer*- und *Abstract Factory*-Muster, für die Aktualisierung der Benutzeroberfläche und das austauschbare *Look & Feel* vorgestellt.

3.1 COCMS Systemarchitektur

Das COCMS ist eine Anwendung, die auf der Zwei-Schicht-Architektur basiert und Benutzer bei der Administration von Konfigurationsdaten der Switches vom Typ DMS-100F unterstützen soll. Eine Zwei-Schicht-Architektur ist eine Variante der Client-/Server-Architektur, bei der die Anwendung direkt über einen Treiber auf die Datenbank zugreift. Der Benutzer kann das COCMS als eigenständige Anwendung auf der Client-Seite starten, gewünschte Anfragen formulieren und diese an den Datenbankserver schicken. Die Kommunikation zwischen Client und Server wird über die von SUN[1] bereitgestellte *JAVA Database Connectivity* realisiert, die vollständig in der Programmiersprache *JAVA* integriert ist. Als Datenbank wird *PostgreSQL* verwendet, da diese bereits auf dem Webserver existiert.

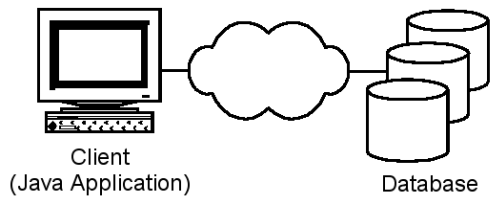


Abbildung 5: Systemarchitektur

[1] Firma SUN Microsystems.

3.2 JAVA Swing

3.2.1 Einführung

Grafische Benutzeroberflächen (engl. GUI, Graphical User Interfaces) stellen einen wichtigen Bestandteil in nahezu allen Software-Produkten dar. Dabei kann die Qualität einer Benutzeroberfläche sehr stark zum Erfolg oder Misserfolg eines Produktes beitragen.

Die bisherigen Benutzeroberflächen von *JAVA*-Programmen konnten sich kaum mit denen von Programmen messen, die ein betriebssystem-spezifisches GUI API (Application Programming Interface) verwendeten. Dieser Rückstand lässt sich auf das Grundkonzept der Plattformunabhängigkeit von *JAVA* zurückführen.

Mit der Einführung des *AWT (Abstract Window Toolkit)* wurde dem Entwickler ein Fundament der Entwicklung grafischer Benutzeroberflächen geliefert.

Da *JAVA* vom Betriebssystem unabhängig sein soll, wurde zu Beginn der Entwicklung des *AWT* nur die grösste gemeinsame Teilmenge von Benutzerschnittstellen-Elementen aller unterstützten Betriebssysteme verwendet. Diese konnten jedoch nicht gerade Begeisterung bei den Anwendern hervorrufen und stellten somit auch den am meisten kritisierten Teil des *JAVA Development Kits* dar. Erst mit der Version 1.2 des *JDK* wurde dem ein Ende bereitet, und es wurde mit den *JAVA Foundation Classes* ein mächtiges Framework geliefert, das die Möglichkeiten der Gestaltung von Benutzeroberflächen mittels dem *AWT* teilweise ersetzt und den Umfang wesentlich erweitert. *Swing* definiert eine beliebige Obermenge von Benutzerschnittstellen-Elementen, ohne lokale Implementierungen berücksichtigen oder verwenden zu müssen [Meyer98].

In diesem Kapitel werden die Grundlagen der professionellen Benutzeroberflächengestaltung mit *JAVA Swing* näher erläutert. Ziel ist es jedoch nicht, die *JAVA Swing*-Klassen vollständig abzuhandeln. Hierfür wird auf die im Anhang dieser Arbeit aufgeführte Literatur verwiesen.

3.2.2 Aufbau der JAVA Foundation Classes

Oft werden die Begriffe „*JAVA Swing*“ und „*JAVA Foundation Classes*“ synonym verwendet. Dies ist jedoch nicht ganz korrekt, da *JAVA Swing* lediglich ein Bestandteil der *JAVA Foundation Classes* darstellt.
In der folgenden Abbildung ist der Aufbau der *JAVA Foundation Classes* näher veranschaulicht:

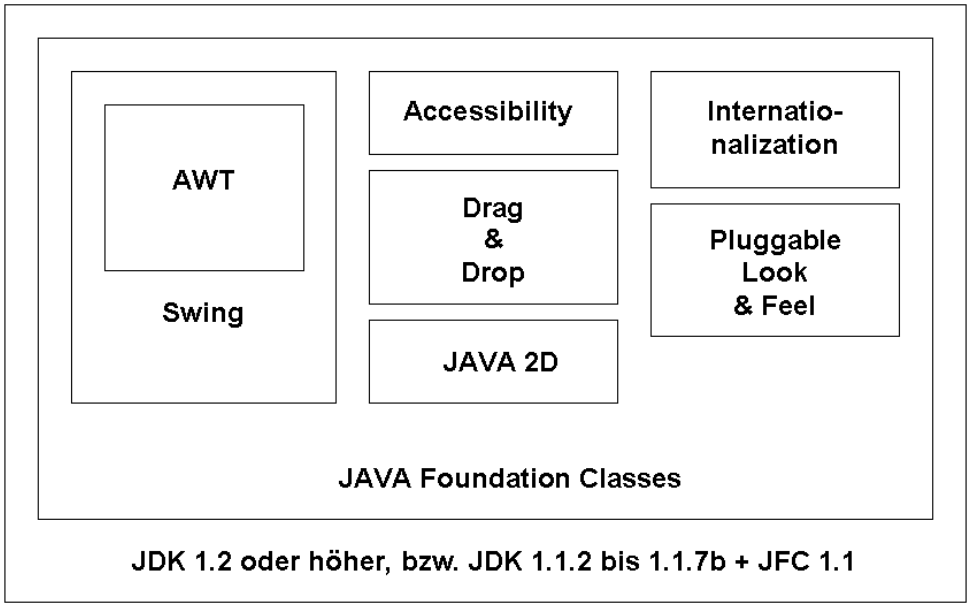


Abbildung 6: Aufbau der JAVA Foundation Classes

Vorstehende Abbildung zeigt das *JAVA Development Kit*, die *JAVA Foundation Classes* und deren Elemente *AWT*, *Swing*, *Accessibility*, *Drag & Drop*, *JAVA 2D*, *Internationalization* sowie *Pluggable Look & Feel* [Eckstein98].
Mit den *JAVA Foundation Classes* wird eine Menge von Klassen bereitgestellt, die das Standard-*JAVA*-Framework um diverse Funktionalitäten erweitern.
Wird das *JAVA Development Kit 1.2* oder höher verwendet, so sind die *JAVA Foundation Classes* bereits im *Development Kit* integriert und man spricht bei dieser Version von “*JAVA2*”. Bei Verwendung der *JAVA Development Kits 1.1.2 bis 1.1.7b* ist es erforderlich, die *JAVA Foundation Classes* in Form einer separaten Klassenbibliothek *(JFC 1.1)* zu installieren.
Im Folgenden wird auf die einzelnen Bestandteile der *JFC* näher eingegangen.

3.2.2.1 Abstract Window Toolkit (AWT)

Eine Programmiersprache, die sich zum Ziel setzt, plattformunabhängige Softwareentwicklung zu unterstützen, muss auch entsprechende Bibliotheken für die grafische Benutzeroberflächengestaltung bereitstellen.
Die Bibliothek muss hierzu im Wesentlichen drei Dinge abdecken:

- Sie muss grafische Interaktionskomponenten (Widgets), wie Fenster, Schaltflächen, Textfelder, Menüs und Container unterstützen.
- Sie muss grafische Primitivoperationen wie Linien und Polygone zeichnen, Farben und Zeichensätze zuweisen können.
- Sie muss ein Modell zur Behandlung der Ereignisse definieren.

Aufgrund der Portabilität von *JAVA*-Programmen kann schnell die Problematik auftreten, dass eine Komponente oder ein bestimmtes Feature einer Applikation von einer bestimmten Plattform nicht unterstützt wird. Die Bibliothek darf daher nur die Komponenten beinhalten, die von jeder grafischen Oberfläche unterstützt werden. Hierfür wurde in *JAVA* das *Abstract Window Toolkit* definiert, das Oberflächenelemente konkreter Plattformen wie *Windows*, *MacOS* oder *UNIX* implementiert und jede Komponente über Zwischenklassen, sogenannte Peer-Klassen, auf eine Komponente der Plattform abbildet. Hieraus resultiert, dass portierte Anwendungen auf dem jeweiligen Rechner ein plattformspezifisches Aussehen erhalten. Des Weiteren ist das *AWT* aufgrund seiner begrenzten Anzahl von Oberflächenelementen sehr einfach gehalten, so dass die Gestaltung einer professionellen Oberfläche nur mühevoll realisierbar ist.
Dies veranlasste SUN, die *Swing*-Klassen parallel zu den *AWT*-Klassen in die *JAVA Foundation Classes* aufzunehmen.

3.2.2.2 Swing

Swing erweitert das *AWT* an einem Ast der *AWT*-Hierarchie. Die Ausgangsklasse stellt hierbei die Klasse *java.awt.Container* dar. Das erste Element in der *Swing*-Hierarchie ist die von *java.awt.Container* abgeleitete Klasse *JComponent*, welche an eine Reihe von Benutzerschnittstellen-Elemente vererbt wird, die die vorhandenen *AWT*-Elemente erweitern bzw. ersetzen. Die folgenden Abbildungen veranschaulichen dies näher:

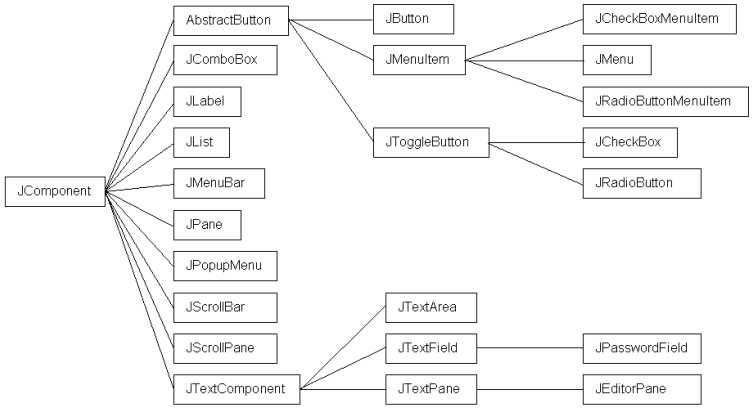


Abbildung 7: Swing-Komponenten, die die AWT-Komponenten ersetzen

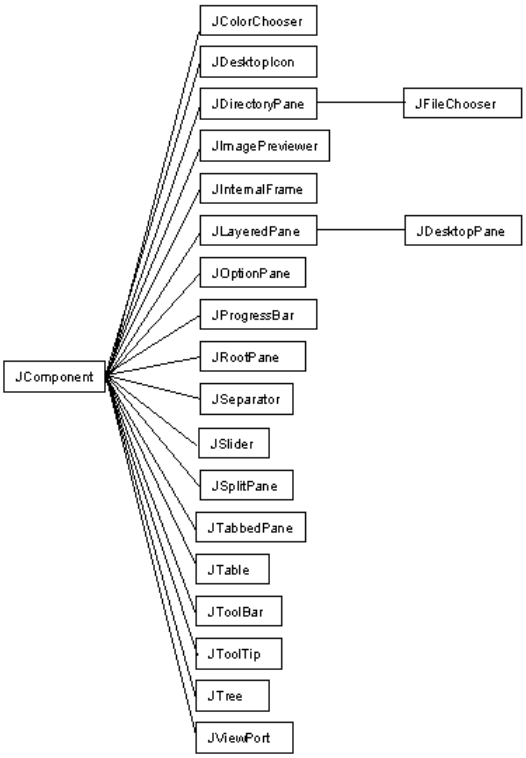


Abbildung 8: Swing-Komponenten, die die AWT-Komponenten erweitern

Im Gegensatz zu *AWT* setzt *Swing* die Existenz von Betriebssystemelementen nicht voraus. Alle Elemente, die zur Darstellung der Benutzeroberfläche benötigt werden, werden von *JAVA* selbst gezeichnet und greifen nicht auf Peer-Klassen zurück. Dieser Ansatz und das in Kapitel 3.2.4 erklärte *MVC*-Konzept (*Model-View-Controller*) sind Voraussetzungen dafür, dass *Swing*-Komponenten ihre Gestalt und Form (*Look & Feel*) beliebig adaptieren können.
Des Weiteren werden *AWT*-Komponenten als schwergewichtig (heavyweight components) bezeichnet, da sie indirekt vom Betriebssystem kontrolliert werden.
Swing-Komponenten hingegen gelten als leichtgewichtig (lightweight components), da sie nur mit dem *JAVA*-Framework arbeiten und nicht auf Betriebssystem-funktionen zurückgreifen.

3.2.2.3 Accessibility

Accessibility steht für die Anpassungsfähigkeit einer Benutzerschnittstelle, so dass diese beispielsweise für Personen mit Behinderungen angepasst werden kann.
Ein Beispiel hierfür ist der Einsatz von Ein- und Ausgabegeräten wie zum Beispiel Spracheingabe oder Sprachausgabe.

3.2.2.4 Drag & Drop

Das Verschieben von Daten mittels „Ziehen“ und „Fallenlassen“ (*Drag & Drop*) ermöglicht den einfach und effizienten Datenaustausch zwischen verschiedenen Anwendungen. Das *Drag & Drop*-API erlaubt den Einsatz dieser Methode sowohl zwischen *JAVA*-Anwendungen als auch mit anderen betriebssystem-spezifischen Anwendungen.

3.2.2.5 JAVA 2D

Mit dem *JAVA 2D*-API werden die bestehenden Möglichkeiten der Bildverarbeitung stark erweitert. Neue Klassen und Funktionen befinden sich in den Paketen

- *java.awt* → Benutzerschnittstellen-Elemente und Zeichenfunktionen.
- *java.awt.image* → Bildverarbeitung.
- *java.awt.color* → Farben.
- *java.awt.font* → Schriften.
- *java.awt.geom* → geometrische Formen und Kurven.

JAVA 2D ist seit dem *JAVA Development Kit 1.2* verfügbar.

3.2.2.6 Internationalization

Seit *JDK 1.1* bietet *JAVA* die Möglichkeit der Internationalisierung (engl.: *Internationalization*) von Anwendungen. Mit dieser Technik, die auch unter dem Namen „Lokalisierung“ bekannt ist, ist es möglich, Anwendungen zu entwickeln, die in vielen Ländern und Regionen der Welt verwendbar sind. Hierbei werden die sprach- und kulturspezifischen Teile von einer Anwendung getrennt und ausgelagert.

3.2.2.7 Pluggable Look & Feel (PLAF)

Das *Pluggable Look & Feel*-Konzept ist seit *JDK 1.2* verfügbar und ermöglicht es, das Aussehen einer Oberfläche zu wechseln, ohne Änderungen am Quellcode vornehmen zu müssen. Der Wechsel kann hierbei sogar zur Laufzeit der Anwendung stattfinden. Hiermit wird die Möglichkeit geschaffen, Anwendungen zu entwickeln, die ein *Windows*-, *Motif*-, *Macintosh*- oder *JAVA(Metal)*-Aussehen haben.
Des Weiteren lassen sich ebenfalls eigene *Look & Feels* entwickeln, die z.B. die Adaption einer Anwendung an ein Corporate Identity zulassen.

3.2.3 Swing im Vergleich zu AWT

Auf den vorherigen Seiten wurden bereits die Unterschiede des *AWT* gegenüber *Swing* angedeutet. An dieser Stelle sollen diese Ausführungen nun vertieft werden.

3.2.3.1 Leichtgewichtige Komponenten (Swing)

Nicht ressourcen-intensiv

Aufgrund des Ansatzes, Benutzeroberflächen-Elemente vollständig auf *JAVA*-Ebene zu zeichnen und keine Peer-Klassen sowie Betriebssystemklassen zu verwenden, ist der Einsatz von *Swing*-Komponenten deutlich weniger ressourcen-intensiv als der von *AWT*-Komponenten.

Plattformunabhängigkeit

Mittels der vollständigen Implementierung der *Swing*-Komponenten in *JAVA* ist das *Look & Feel* plattformunabhängig und ermöglicht es, während der Laufzeit einer Anwendung das *Look & Feel* zu wechseln.

JAVA-*Swing*-Komponenten arbeiten allesamt nach dem *Model*-*View*-*Controller*-Konzept.

Die nachfolgende Abbildung veranschaulicht den Unterschied zwischen leicht- und schwergewichtigen Komponenten bezüglich deren Struktur:

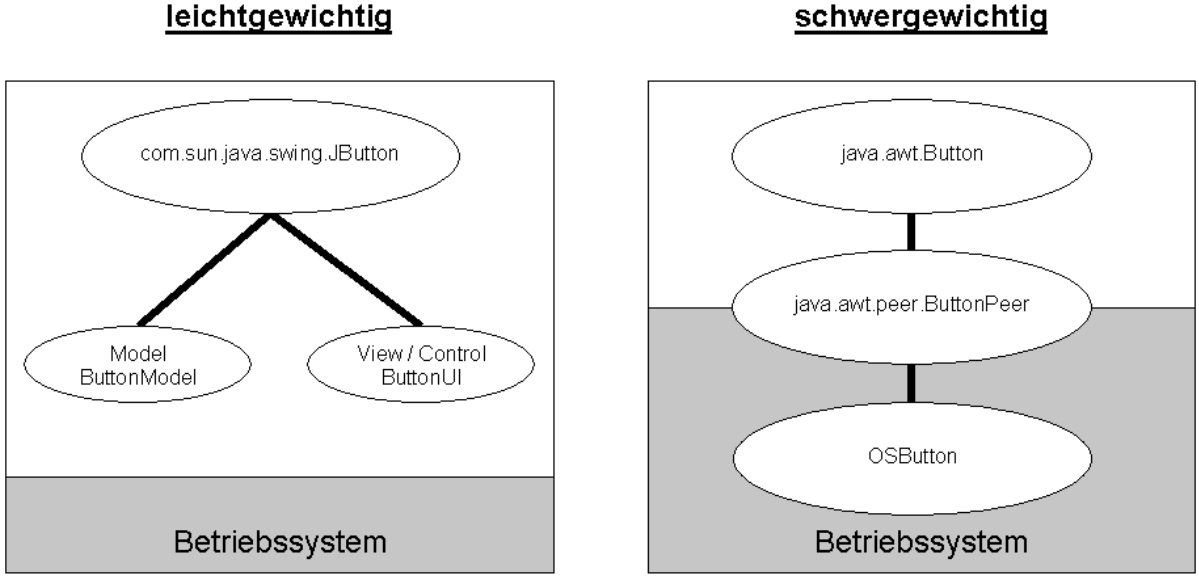


Abbildung 9: Schwergewichtige und leichtgewichtige Komponenten

Der rechte Teil der Abbildung zeigt die Verwendung einer schwergewichtigen Komponente. Zwischen Betriebssystem und *JAVA*-Umgebung befindet sich die Peer-Klasse *java.awt.peer.ButtonPeer*, welche die Schnittstelle zum Betriebssystem darstellt. Im linken Teil der Abbildung wird die Verwendung einer leichtgewichtigen Komponente näher veranschaulicht. Unschwer lässt sich erkennen, dass die Komponente vollständig in *JAVA* implementiert und unabhängig vom Betriebssystem ist. Zudem lässt sich der Abbildung entnehmen, dass hier ein *Model-View-Controller*-Konzept zugrunde liegt.

Generell lassen sich *Swing*-Klassen durch ein führendes *„J“* im Namen erkennen.

Eine simultane Verwendung von *Swing*- und *AWT*-Klassen ist ebenfalls möglich, hierbei können jedoch Probleme auftreten. Beispielsweise muss beim Auftreten eines Ereignisses dieses explizit an eine Oberklasse übergeben werden, wenn das Ereignis von einer schwergewichtigen Komponente an eine leichtgewichtige Komponente weitergeleitet werden soll. Des Weiteren werden bei der gemeinsamen Darstellung die leichtgewichtigen Komponenten von den schwergewichtigen überdeckt.

Transparenz

Mit *Swing* lassen sich beliebige Grundflächen (z.B. Kreise, Ovale, usw.) darstellen und auch überlappen, ohne dass eine Komponente die andere überdeckt (Transparenz). Dies ist mit dem *AWT* nicht möglich.

3.2.3.2 Schwergewichtige Komponenten (AWT)

Ressourcen-intensiv

Im Gegensatz zu *Swing*-Komponenten erscheinen *AWT*-Komponenten oft etwas träge in ihrer Reaktion. Dies ist darauf zurückzuführen, dass neben der eigentlichen *JAVA*-Komponentenklasse noch eine betriebssystem-spezifische Klasse sowie eine Zwischenklasse (Peer-Klasse) in den Datenaustausch involviert ist.
Erfolgt beispielsweise eine Anforderung für das Neuzeichnen einer Komponente, so wird diese von der *JAVA*-Komponentenklasse an die Peer-Klasse und anschliessend von der Peer-Klasse an das Betriebssystem weitergeleitet. Dies hat eine wesentlich höhere Beanspruchung der Ressourcen zur Folge.

Plattformabhängigkeit

Aufgrund der Einbeziehung einer betriebssystem-spezifischen Klasse in die Darstellung einer Komponente gelten *AWT*-Komponenten als plattformabhängig. Dies hat zur Folge, dass eine Anwendung ein nicht einheitliches Erscheinungsbild erhält und somit auf jeder Plattform unterschiedlich interpretiert wird. Die eigentliche Problematik liegt jedoch viel mehr in der unterschiedlichen Leistungsfähigkeit der grafischen Darstellung verschiedener Betriebssysteme. Eine *AWT*-Implementierung kann somit nur die „Schnittmenge“ verschiedener Systeme nutzen. Bestimmte Features, die nur von einem bestimmten System zur Verfügung gestellt werden, können nicht genutzt werden.

Keine Transparenz

AWT-Komponenten sind nicht transparent und ermöglichen lediglich die Darstellung einer komplett rechteckigen Fläche. Soll ein rundes oder ovales Objekt mittels *AWT* dargestellt werden, so muss immer berücksichtigt werden, dass dieses Objekt von einem Rechteck umgeben ist.

3.2.4 Architektur – Das Model-View-Controller-Konzept (MVC)

Swing-Komponenten basieren auf der *Model-View-Controller*-Architektur.
Das Grundprinzip dieser Architektur besteht in der Trennung der Daten von ihrer Darstellung, d.h. die Daten werden zentral und unabhängig von ihrer grafischen Repräsentation gespeichert, wobei diese Datenbasis als sogenanntes *Model* bezeichnet wird. Die grafische Repräsentation wird von der *View* einer Komponente in Form einer Benutzeroberfläche übernommen, die die Daten aus dem *Model* bezieht und nicht selbst speichert. Der *Controller* ist für die Interaktion mit dem Anwender verantwortlich, d.h. er erhält Eingaben vom Benutzer über eine Maus oder Tastatur, leitet diese an das *Model* weiter und sendet gegebenenfalls eine Änderungsnachricht an die *View*.

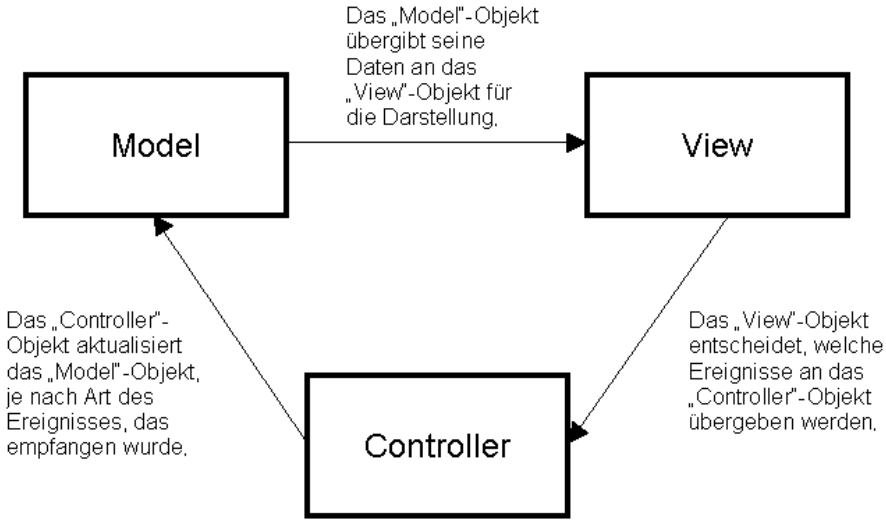


Abbildung 10: Model-View-Controller-Konzept

Ein wichtiger Aspekt hierbei ist, dass einem *Model* mehrere *Views* gleichzeitig zugewiesen werden können. Um Veränderungen des *Models* in allen *Views* sichtbar zu machen, wird ein Benachrichtigungsmechanismus implementiert, mit dem das *Model* die zugeordneten *Views* über Änderungen informiert. Diese Vorgehensweise entspricht dem Entwurfsmuster *Observer*, wie es in Kapitel 3.5.2 näher erläutert wird.

Die Verwendung der *Model-View-Controller*-Architektur verzeichnet folgende Vorteile:

- Das Datenmodell muss sich nicht um seine Repräsentation kümmern.
- Verkürzte Entwurfs- und Erstellungszeit durch Wiederverwendbarkeit einzelner Komponenten der Architektur.
- Reduzierter Programmieraufwand durch Separation des Systems in mehrere Teile
  → Ermöglichung der separaten Bearbeitung bei Problemen oder Änderungen.
- Bindung zwischen *Model* und *View* kann dynamisch zur Laufzeit geschehen (z.B. Änderung des *Look & Feel*s einer Anwendung zur Laufzeit).
- Sicherung der Flexibilität einer Anwendung durch Trennung der Darstellung von der eigentlichen Datenbasis (z.B. Bearbeitung eines Dokuments ohne eine bestimmte Ansicht zu öffnen).
- Effizientere Mehrfachdarstellung der Datenbasis möglich, da Daten nur einmalig in zentraler Form vorliegen.

3.2.5 Architektur – Das Model-Delegate-Konzept

Bei der Umsetzung des *MVC*-Musters in der Praxis hat es sich jedoch herausgestellt, dass *View* und *Controller* meist stark voneinander abhängig sind. Zu je einer *View* muss genau ein *Controller* existieren. Deshalb wird sehr häufig eine leicht abweichende Version des *MVC*-Musters verwendet, das sogenannte *Model-Delegate*-Konzept, welches bei den *Swing*-Objekten zum Einsatz kommt. In diesem Konzept, auch als *Separable-Model-Architecture (SMA)* bekannt, werden *View* und *Controller* in einem *UI-Delegate (User Interface)* zusammengefasst, der die Komponenten zeichnet und die Ereignisse der Benutzeroberfläche behandelt. Die Kommunikation erfolgt hierbei in beiden Richtungen.

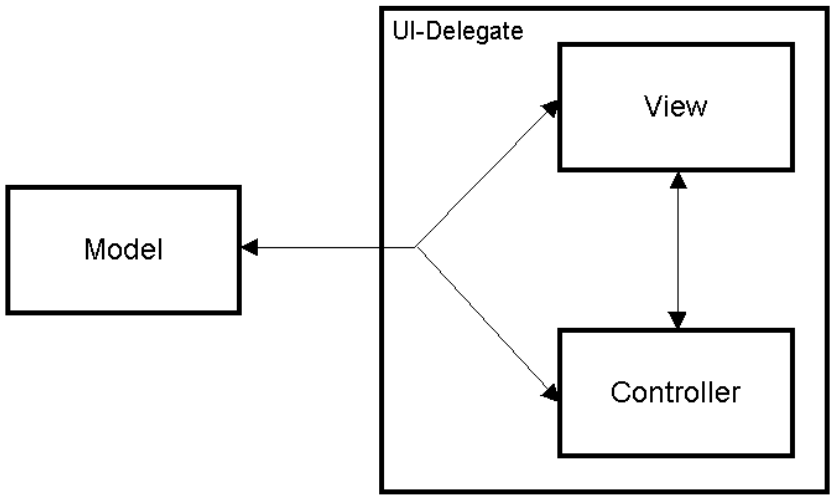


Abbildung 11: Separable-Model-Architecture

Das *Model*-Objekt enthält Informationen über den aktuellen Zustand der Komponente, der *UI-Delegate* hingegen ist für die Repräsentation und Steuerung der Informationen zuständig. Diese Trennung bietet eine Reihe von Vorteilen, wie z.B. die Verbindung mehrerer Darstellungen mit nur einem *Model*-Objekt. Bei einer Änderung des *Models* werden automatisch alle Darstellungen aktualisiert.
Dieses Prinzip kommt beispielsweise beim *Pluggable Look & Feel* zum Einsatz, bei dem sich mit dem Umschalten des *Look & Feel*s alle Darstellungsobjekte in ihrem Aussehen automatisch ändern.

Der Einsatz der *Separable-Model-Architecture* wird anhand der *Swing*-Komponente *JButton* in der folgenden Abbildung näher veranschaulicht:

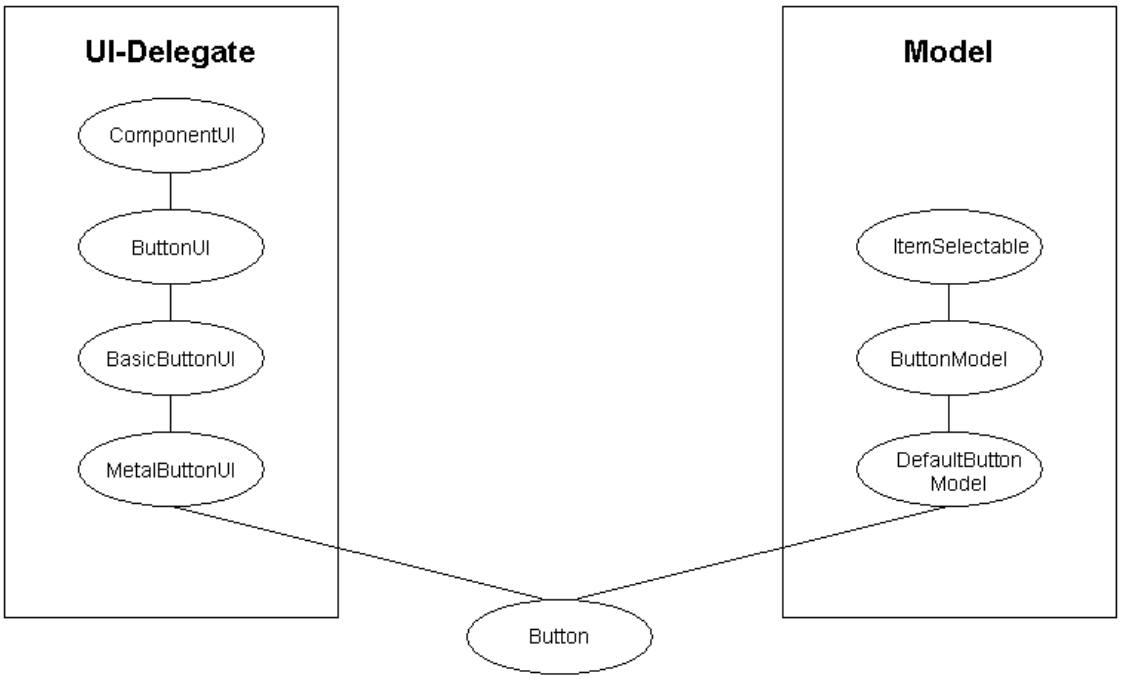


Abbildung 12: Beispiel des Einsatzes der Separable Model Architecture bei der Swing-Komponente JButton

3.2.6 Zusammenfassung und Ausblick

Generell ist die Verwendung von *Swing* bzw. *AWT* daran auszumachen, welcher Zweck mit der grafischen Darstellung verfolgt wird. Beispielsweise ist bei der Implementierung eines *JAVA-Applets* für das Internet auf dessen Grösse zu achten, damit keine unnötig langen Ladezeiten verursacht werden. An dieser Stelle empfiehlt sich der Einsatz von *AWT*, da *AWT*-Komponenten im Gegensatz zu *Swing*-Komponenten wesentlich kleiner sind und somit beachtlich schneller geladen werden. Hingegen macht die Verwendung von *Swing*-Komponenten bei Applikationen, die lokal auf einer Maschine interpretiert werden, durchaus Sinn, da diese in ihrem Umfang wesentlich mehr Möglichkeiten bieten, professionelle Benutzeroberflächen zu gestalten. In Bezug auf die Performance eines Systems ist ebenfalls abzuwägen, welches Toolkit zum Einsatz kommen soll. Zwar ist *Swing* bei Weitem weniger ressourcen-intensiv wie *AWT*, da weniger Instanzen durchlaufen werden müssen, die Betriebssystem-Methoden, wie sie bei *AWT* für das Zeichnen von Objekten verwendet werden, zeichnen jedoch deutlich schneller.

Swing bietet dem Entwickler im Gegensatz zu *AWT* deutlich mehr Flexibilität sowie die Möglichkeit, vorhandene Komponenten zu erweitern und an die eigenen Datenstrukturen anzupassen. Des Weiteren existieren eine ganze Reihe vorgefertigter Komponenten, wie zum Beispiel Tabellen, Bäume oder Standard-Dialoge, die sehr einfach zu nutzen und für die meisten Bedürfnisse vollkommen ausreichend sind.

Somit ist unter *Swing* der Weg zur gewünschten Benutzeroberfläche sehr viel einfacher als unter Verwendung des *AWT*.

3.3 Datenmodellierung

3.3.1 Einführung

Datenbanken stellen ein Mittel dar, um Daten in strukturierter Form zu speichern.
Sie werden meist von Applikationen in Anspruch genommen und sind Teil eines umfangreichen Informationssystems, welches von der Datenbank Informationen anfordert, auswertet, und Daten zur Speicherung übergibt.
Für die Verarbeitung von Daten in EDV-Systemen wird als Strukturierungsmodell die Tabelle verwendet, da sie sich mathematisch durch Matrizen beschreiben und durch iterative Algorithmen auswerten lässt [Kemper01].

3.3.2 Das Konzept relationaler Datenbanksysteme

Tabellen werden im relationalen Datenmodell als Relationen bezeichnet. Eine Relation besteht in ihrer intentionalen Ausdehnung aus einer festen Anzahl von Attributen (Spalten) und in extensionaler Ausdehnung aus einer variablen Anzahl von Tupeln (Zeilen). Der Wertebereich eines Attributes wird als Domäne bezeichnet und entspricht somit etwa einem elementaren Datentyp in einer Programmiersprache.
Relationen weisen die folgenden Eigenschaften auf:

- Jeder Eintrag in einer Relation repräsentiert einen Datenwert, der atomar ist, d.h. es gibt keine Wiederholungsgruppen.
- Eine Spalte ist homogen, d.h. alle Werte einer Spalte entstammen derselben Domäne.
- Jeder Spalte ist ein Attribut mit eindeutigem Namen zugewiesen.
- Alle Tupel einer Relation sind voneinander verschieden.
- Zeilen- und Spaltenreihenfolgen innerhalb einer Relation sind zu jedem Zeitpunkt beliebig und wirken sich nicht auf darauf zugreifende Funktionen aus.

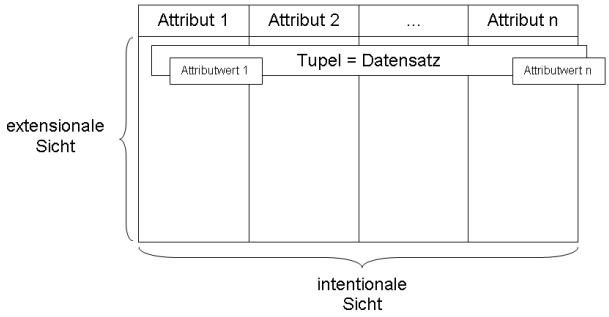


Abbildung 13: Aufbau einer Relation

Jede Relation besitzt genau ein Attribut, das als Primärschlüssel definiert ist. Dieses hat die Aufgabe, jedes Tupel innerhalb der Relation eindeutig zu identifizieren und darf somit somit keine Nullwerte enthalten. Der Primärschlüssel kann ebenfalls aus einer Gruppe von Attributen zusammengesetzt sein, hierbei wird von einem zusammengesetzten Schlüssel gesprochen. Die identifizierende Eigenschaft des Primärschlüssels wird als funktionale Abhängigkeit der Nicht-Schlüssel-Attribute vom Primärschlüssel bezeichnet.
Des Weiteren können in einer Relation Fremdschlüssel existieren. Diese sind dadurch gekennzeichnet, dass sie nur Werte der Primärschlüssel referenzierter Relationen beinhalten können. Sie dienen der Referenzierung auf andere Relationen.
Weitere Schlüsselarten, die in einer Relation existieren können, sind:

- **Schlüsselkandidaten:**
  Ein Schlüsselkandidat hat die gleiche Eigenschaften wie ein Primärschlüssel und kann dessen Rolle übernehmen.

- **Zweitschlüssel:**
  Ein Zweitschlüssel kann jedes beliebige Attribut ausser dem Primärschlüssel sein, dieser sorgt jedoch nicht für eine eindeutige Identifizierung des Tupels.

Generell werden Relationennamen mit vorangestelltem *R*. gekennzeichnet. Die Attributnamen werden in Klammern nachgestellt, der Primärschlüssel wird unterstrichen dargestellt.

R.Relationenname (<u>Attribut 1</u>, Attribut 2, ..., Attribut n)

3.3.3 Das Normalisierungskonzept

Das Ablegen von Daten in einer Relation kann sehr ineffizient sein, wenn die Strukturierung eine gewisse Redundanz der Daten zulässt. Redundanz von Daten bedeutet die mehrfache Speicherung gleicher Daten in einer Relation.
Die folgende Tabelle veranschaulicht dies beispielhaft:

Name	**Strasse**	**Ort**	**Buchtitel**	**Leihdatum**
Heller	Richentalstrasse	Konstanz	JAVA lernen	27.08.02
Sütterlin	Rheingutstrasse	Konstanz	Baustatik	24.07.02
Heller	Richentalstrasse	Konstanz	PERL5	26.08.02
Steppacher	Kindlebildstrasse	Konstanz	Sicherh. im Netz	21.08.02
Ernsberger	Blarerstrasse	Konstanz	PostgreSQL	23.08.02

Tabelle 1: Beispiel für redundande Datenrepräsentation

Dieser vorliegende Zustand wird auch als erste Normalform bezeichnet, da die Spalten nur atomare Werte, d.h. einfache voneinander unabhängige Werte, enthalten. Redundanz in einer Datenbank führt zu erhöhtem Speicherbedarf und dem Auftreten von Inkonsistenzen der gespeicherten Daten, wenn z.B. nicht alle Einträge beim Zugriff synchronisiert werden.
Mit dem relationalen Datenbanksystemmodell wird das Ziel verfolgt, diese Redundanz zu minimieren bzw. zu unterbinden. Das Grundprinzip besteht darin, die Datenbank aus mehreren Relationen aufzubauen, die aufeinander referenzieren.
Dieser Vorgang wird auch Normalisierung genannt und ist beispielhaft anhand der folgenden Relationen veranschaulicht.

Für das vorliegende Beispiel ergeben sich somit die folgenden Relationen:

Kunden-Nr.	Name	Strasse	Ort
1	Heller	Richentalstrasse	Konstanz
2	Sütterlin	Rheingutstrasse	Konstanz
3	Steppacher	Kindlebildstrasse	Konstanz
4	Ernsberger	Blarerstrasse	Konstanz

Tabelle 2: Beispiel für redundanzfreie Datenrepräsentation → Tabelle Kunden

Buch-Nr.	Buchtitel
1	JAVA lernen
2	Baustatik
3	PERL5
4	Sicherh. Im Netz
5	PostgreSQL

Tabelle 3: Beispiel für redundanzfreie Datenrepräsentation → Tabelle Bücher

Kunden-Nr.	Buch-Nr.	Leihdatum
1	1	27.08.02
2	2	24.07.02
1	3	26.08.02
3	4	21.08.02
4	5	23.08.02

Tabelle 4: Beispiel für redundanzfreie Datenrepräsentation → Tabelle Leihvorgänge

Bezüglich der Normalisierung ist zu nennen, dass insgesamt fünf Normalformen existieren, die hierarchisch aufeinander aufbauen. Auf deren Details soll jedoch an dieser Stelle nicht näher eingegangen werden.

Das Beispiel zeigt, dass die zu Beginn vorhandene Redundanz mit der Erstellung von Relationen, die aufeinander referenzieren, beseitigt wird. Keine der Relationen weist in irgendeinem Bereich identische Werte auf.
In der Relation „Leihvorgänge“ wird auf die Relationen „Kunde“ und „Bücher“ referenziert. Hierbei ist eine eindeutige Zuordnung zwischen Kunden, Büchern und Leihvorgängen möglich. Des Weiteren bietet dies den Vorteil, dass in der Praxis nicht bei jedem Vorgang der Speicherung sämtliche Daten neu eingegeben werden müssen. Lediglich die noch nicht in den Relationen existierenden Einträge werden ergänzt. In den Relationen „Kunden“ und „Bücher“ wird jeder Datensatz eindeutig über eine Nummer identifiziert. Dieses eindeutige Identifizierungsmerkmal wird durch den Primärschlüssel dargestellt. Ist der Wert des Primärschlüssels bekannt, so lassen sich alle zugehörigen Werte eines Datensatzes ermitteln.
In der Relation „Leihvorgänge“ hingegen existiert kein Primärschlüssel. Diese Relation enthält lediglich Fremdschlüssel, die auf die beiden anderen Relationen referenzieren. Fremdschlüssel sind dadurch gekennzeichnet, dass sie nur Werte des Primärschlüssels der referenzierten Relation enthalten können. Eine eindeutige Identifizierung ist somit nur mithilfe eines Primärschlüssels möglich. Aus diesem Grund wird recht häufig bei der Modellierung einer Datenbank eine zusätzliche Spalte der Relation hinzugefügt, die es ermöglicht, die Einträge eindeutig zu identifizieren.

Die wesentliche Funktion eines relationalen Datenbanksystems besteht darin, die verschiedenen Relationen einer Datenbank anhand der Schlüsselwerte miteinander zu verknüpfen und aufgrund dieser Verknüpfungen für die Integrität der gespeicherten Daten zu sorgen.

3.3.4 Beziehungstypen

Bezüglich der Verknüpfungen von Relationen lassen sich mehrere Typen unterscheiden:

- **„1:1“-Verknüpfung**
  Zu jedem Datensatz einer Relation existiert exakt ein Datensatz in einer anderen Relation, der auf diesen referenziert. Diese Zuordnung mag eventuell als unnötig erscheinen, da aufgrund der eindeutigen Zuordnung keine Redundanz entstehen kann. Sie wird jedoch verwendet, um durch die Aufteilung in mehrere Relationen eine bessere Übersichtlichkeit zu schaffen.

- **„1:n“-Verknüpfung**
  Zu einem Datensatz einer Relation existieren mehrere Datensätze in einer anderen Relation, die auf diesen referenzieren.

- **„n:m“-Verknüpfung**
  Zu mehreren Datensätzen einer Relation existieren mehrere Datensätze in einer anderen Relation, die auf diese referenzieren. In der Theorie ist dieser Ansatz denkbar, in realen Datenbanksystemen jedoch nicht umsetzbar und würde zu Redundanz führen. Aus diesem Grund wird in der Praxis eine zusätzliche Relation geschaffen und die „n:m“-Beziehung in zwei „1:n“-Beziehungen umgewandelt.

Die folgende Abbildung veranschaulicht die verschiedenen Verknüpfungstypen anhand dem genannten Beispiel:

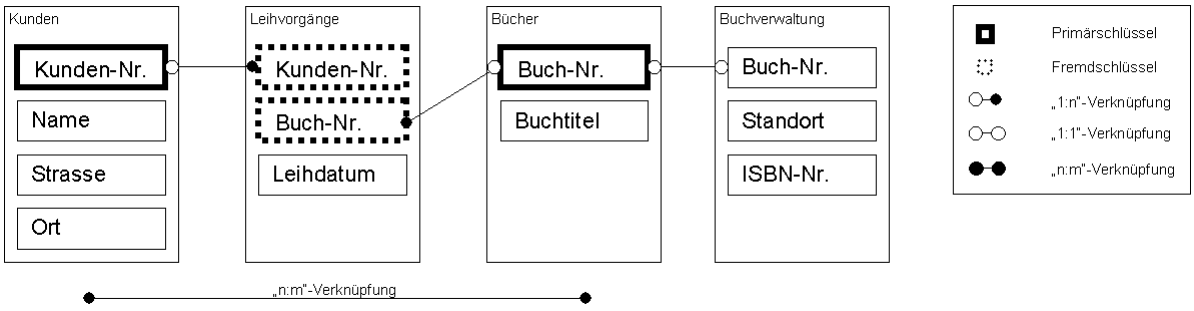


Abbildung 14: Verknüpfungstypen innerhalb der Datenbank

3.3.5 Struktur und Funktionen der Datenbanksprache SQL

SQL (Standard Query Language) stellt eine Standard-Abfragesprache für die Kommunikation mit Datenbanken dar und dient dem Zugriff auf Daten in Datenbanksystemen sowie deren Abfrage, Aktualisierung und Verwaltung.
Generell lässt sich *SQL* in drei Befehlsklassen untergliedern.

3.3.5.1 DDL (Data Definition Language)

Mit diesen Befehlen lassen sich Datenstrukturen (Datenbanken, Relationen, usw.) definieren, bearbeiten und löschen. Hierfür typische Befehle sind *CREATE, ALTER* und *DROP*, die dem Erstellen, Modifizieren und Löschen von Relationen dienen.

3.3.5.2 DML (Data Manipulation Language)

Diese Befehle ermöglichen die Durchführung von Abfragen und Modifikationen von Daten in der Datenbank. Die hierbei wesentlichen Befehle sind *INSERT, SELECT, UPDATE* und *DELETE* und dienen dem Einfügen, Selektieren, Ändern und Löschen von Daten.

3.3.5.3 DCL (Data Control Language)

Diese Befehle erlauben die Vergabe von Zugriffsrechten in einer Datenbank, die von mehreren Personen genutzt wird und können auf diverse Objekte (Relationen, Views, usw.) angewendet werden. Hierzu zählen Befehle wie *GRANT* und *REVOKE*, die dem Benutzer Zugriffsrechte gewähren bzw. entziehen.

Datenbanksysteme sind in der Regel als System eines Serverprozesses realisiert, der eine Datenbank auf der vorliegenden Plattform verwaltet. Der Benutzerzugriff erfolgt meist über das Netzwerk mithilfe eines herstellerspezifischen Anwendungsprotokolls. Um Anwendungsentwicklern die Möglichkeit zu bieten, aus einer Applikation heraus auf das vorhandene Datenbanksystem zuzugreifen, liefert der Hersteller eines Datenbanksystems eine Softwarebibliothek in Form eines Treibers, der den Zugriff auf eine Datenbank über eine Programmiersprache ermöglicht. Des Weiteren existieren auch hersteller- und plattformunabhängige Schnittstellen (*JAVA*), die eine sehr fortschrittliche und elegante Softwareentwicklung möglich machen.

3.4 JAVA Database Connectivity (JDBC)

3.4.1 Einführung

Heutzutage haben Anwendungen, die auf Datenbanken zugreifen, grosse Bedeutung erlangt. Oft stellen Datenbanken die einzige Möglichkeit dar, Benutzern eine gemeinsame Datenbasis zur Verfügung zu stellen. Der Anwender greift jedoch hierbei nicht direkt auf die Daten zu, sondern benutzt bestimmte Schnittstellen, über welche die Daten im Datenbankmanagementsystem abgefragt, eingefügt, modifiziert oder gelöscht werden können.
Aufgrund unterschiedlicher Datenbanksysteme und Schnittstellen ist eine einheitliche Sprache für die Kommunikation zwischen Datenbank und Anwendung erforderlich. Hierfür wurde *SQL* entwickelt. *SQL* stellt keine funktional vollständige Programmiersprache dar, d.h. Programme können nicht direkt in *SQL* programmiert werden. *SQL-Statements* erfordern die „Einbettung“ in ein Programm, das in einer Sprache wie *C* oder *Java* entwickelt wurde.

Seit dem *JDK 1.1* ist die *JAVA Database Connectivity* Bestandteil der Standarddistribution von *JAVA* und bezeichnet ein von SUN entwickeltes Paket, das auf den *JAVA*-Basisklassen aufsetzt und es ermöglicht, aus Applikationen durch Bereitstellung der entsprechenden Objekte und Methoden auf beliebige Datenbanken zuzugreifen.
Mittlerweile ist *JDBC* ein weit geltender Standard und eine Vielzahl von *JDBC*-Treibern stehen im Internet zum Download zur Verfügung.
JDBC setzt auf dem *X/OPEN SQL-Call-Level-Interface (CLI)* auf und besitzt somit die gleiche Basis wie die *ODBC*-Schnittstelle *(Open Database Connectivity).*

3.4.2 Allgemeine Funktionsweise von JDBC

Mittels *JDBC* wird eine Schnittstelle zur Verfügung gestellt, mit deren Unterstützung es möglich ist, *SQL*-Datenbankanfragen zu stellen und diese auszuwerten.
Generell lässt sich eine *JDBC*-Kommunikation in folgende Schritte untergliedern:

- Herstellung einer Verbindung zum Datenbankmanagementsystem über den entsprechenden *JDBC*-Treiber.
- Erstellung eines *Statement*-Objektes.
- Weitergabe des auszuführenden *Statements* über das *Statement*-Objekt an das Datenbankmanagementsystem.
- Abholung der Ergebnisse über die Ergebnisdatensätze *(ResultSets).*
- Bearbeitung der Ergebnisse.
- Schliessen der Verbindung zur Datenbank.

Dabei spielt es keine Rolle, von welchem Hersteller das Datenbank-managementsystem stammt, da *JDBC* eine Vielzahl von Treibern für unterschiedlichste Systeme zur Verfügung stellt.
Existiert kein Treiber für das vorliegende Datenbankmanagementsystem, so besteht die Möglichkeit, einen *ODBC*-Treiber zu verwenden. Ein *ODBC*-Treiber wird von nahezu jedem *DBMS*-Hersteller angeboten. Hierbei kommuniziert *JDBC* mit der *ODBC*-Schnittstelle und kann somit auch für Datenbankmanagementsysteme verwendet werden, für die es keine direkten Treiber gibt.
JDBC ist vollständig in *JAVA* implementiert und kann sehr einfach in *JAVA*-Applikationen eingebunden werden. Somit erbt *JDBC* automatisch alle Vor- und Nachteile von *JAVA*, wie z.B. die hohe Portabilität und die Möglichkeit der Benutzung der Anwendung in Form eines *Applets* aus dem Browser heraus, jedoch auch die geringere Geschwindigkeit.

3.4.3 Open Database Connectivity (ODBC)

ODBC stellt das Pendant zu *JDBC* dar und wird hier nur der Vollständigkeit halber aufgeführt. *ODBC* ist eine Schnittstelle, die von Microsoft entwickelt wurde und Möglichkeiten bereitstellt, auf Datenbanken zuzugreifen. Die Unterstützung von *ODBC* ist nicht auf Microsoft beschränkt. Fast alle *DBMS*-Hersteller unterstützen *ODBC*, unter anderem Oracle, Informix, Novell, Borland, IBM, usw. Bei der Kommunikation mit dem Datenbankmanagementsystem werden *SQL*-Befehle verwendet, die in *ODBC*-Anweisungen eingepackt sind. Diese werden jedoch nicht von *ODBC* interpretiert und unterliegen auch keiner Syntaxkontrolle.
Die *SQL*-Befehle werden zu einem *ODBC*-Treiber geschickt, der vom jeweiligen Hersteller des Datenbankmanagementsystems zur Verfügung gestellt wird.
Dieser nimmt die *SQL*-Befehle entgegen und sendet sie mit einem proprietären Protokoll an die Datenbank. Anschliessend nimmt der Treiber die Antworten der Datenbank entgegen und übergibt diese wieder der Anwendung.

3.4.4 Architektur

Generell existieren zwei verschiedene Architekturen, um mittels *JDBC* eine Verbindung zwischen einer Anwendung und einer Datenbank herzustellen.
In der Regel sind dies die *Two-Tier*- und die *Three-Tier*-Architektur, die in der folgenden Abbildung vorgestellt werden:

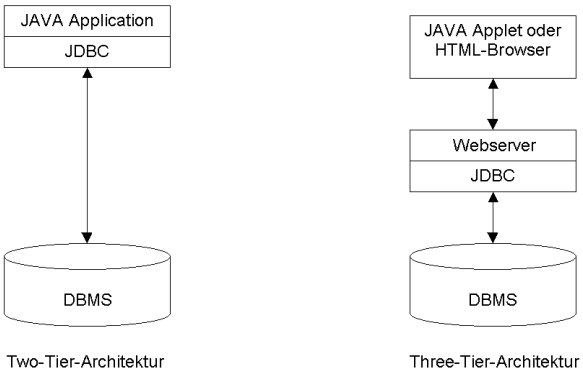


Abbildung 15: JDBC-Architekturen

Der Einsatz der jeweiligen Architektur hängt sehr stark von den Anforderungen der Anwendung bzw. der Hard- und Software-Umgebung des Einsatzgebietes von *JDBC* sowie der Art des zur Verfügung stehenden Treibers ab.

3.4.4.1 Two-Tier-Architektur

Bei Verwendung der *Two-Tier*-Architektur wird durch eine Anwendung mithilfe eines Datenbanktreibers direkt eine Verbindung zum Datenbankmanagementsystem aufgebaut. Diese Architektur entspricht der klassischen Client-/Server-Architektur.
Beispielhaft für diese Architektur ist das Laden eines *JAVA-Applets* mithilfe eines Browsers auf den Client. Das *JAVA-Applet* kommuniziert direkt mit dem Datenbankserver und kann aufgrund der strengen Sicherheitsvorschriften für *Applets* nur mit dem Rechner in Verbindung treten, von welchem das *Applet* geladen wurde. Voraussetzung für eine Kommunikation zwischen *Applet* und Datenbank ist somit, dass die Datenbank und der Webserver sich auf demselben Rechner befinden.
Bezüglich der *Two-Tier*-Architektur existieren zwei Varianten:

- Das *JAVA-Applet* wird von einem Webserver auf einen Client geladen, der sich anschliessend über ein *DBMS*-spezifisches Protokoll mit der Datenbank verbindet. Bei dieser Variante liegt ein sogenannter „Pure *JAVA*“-Treiber vor, der mit dem Laden des *Applets* vom Webserver geladen wird.
- Das *JAVA-Applet* wird von einem Webserver auf einen Client geladen, der sich anschliessend über einen nativen *DBMS*-Treiber mit der Datenbank verbindet.

Beide oben genannte Varianten der *Two-Tier*-Architektur werden in Kapitel 3.4.5.2 und 3.4.5.4 genauer beschrieben.
Die *Two-Tier*-Architektur setzt eine sehr leistungsfähige Maschine voraus, da sich Webserver und Datenbankserver auf demselben Rechner befinden müssen. Zudem muss bei Verwendung eines „Pure *JAVA*“-Treibers dieser bei jeder Verbindung erneut vom Webserver geladen werden, was eine erhöhte Ladezeit des *Applets* zur Folge hat, dafür jedoch die Portabilität verbessert.
Wird anstelle des „Pure *Java*“-Treibers ein nativer Treiber verwendet, so bringt dies zwar einen gewissen Geschwindigkeitsvorteil, da sich der Treiber lokal auf dem System befindet und nicht bei jedem Verbindungsaufbau erneut geladen werden muss, die Portabilität und Softwareverteilung werden jedoch damit extrem eingeschränkt.

3.4.4.2 Three-Tier-Architektur

Die *Three-Tier*-Architektur kommt zum Einsatz, wenn sich Webserver und Datenbankserver auf getrennten Rechnern befinden bzw. mehrere Datenbankserver existieren, zu denen Verbindungen aufgebaut werden sollen.

In diesem Fall ist ein Koppelelement, ein sogenannter *Middle-Tier*-Server erforderlich, der als Gateway zu den Datenbankservern dient.

Bei diesem System wird das *Applet* vom Webserver geladen, welches sich über ein *DBMS*-unabhängiges Protokoll mit dem Gateway auf dem Webserver verbindet. Das Gateway fungiert als Datenbank-Client, übernimmt die Abfragen der Datenbank für das *Applet* und sendet anschliessend die Ergebnisse an das *Applet* zurück.

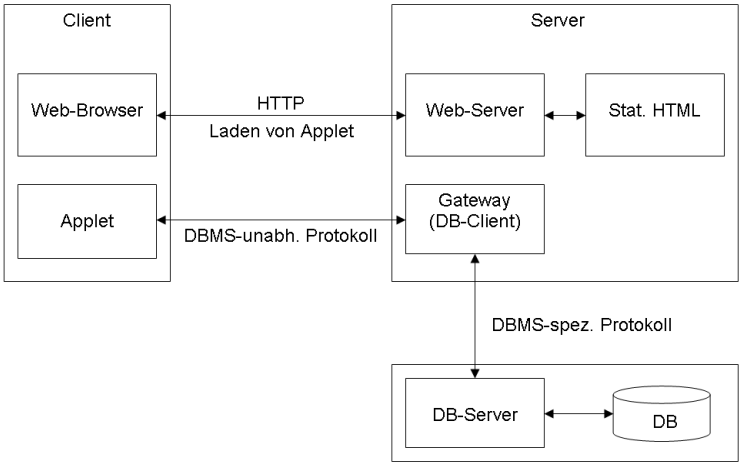


Abbildung 16: Three-Tier-Architektur

Die Trennung des Webservers vom Datenbankserver sorgt für eine erhebliche Entlastung und führt zu erhöhter Sicherheit des Datenbankservers, da der Zugriff nur über das Gateway möglich ist. Die Programmierung des Gateways ist jedoch erheblich komplexer, da eine Umsetzung der *DBMS*-unabhängigen Anweisungen in *DBMS*-spezifische Befehle notwendig ist.

3.4.5 JDBC-Treiber-Typen

Die Verwendung von *JDBC* erfordert den Einsatz von entsprechenden Treibern für die jeweils vorliegende Datenbank. Die Verwaltung der Datenbankverbindungen wird vom *JDBC*-Treiber-Manager übernommen, welcher mehrere Verbindungen gleichzeitig zu unterschiedlichen Datenbanken verwalten kann und somit auch den Zugriff auf verteilte Datenbanken ermöglicht.

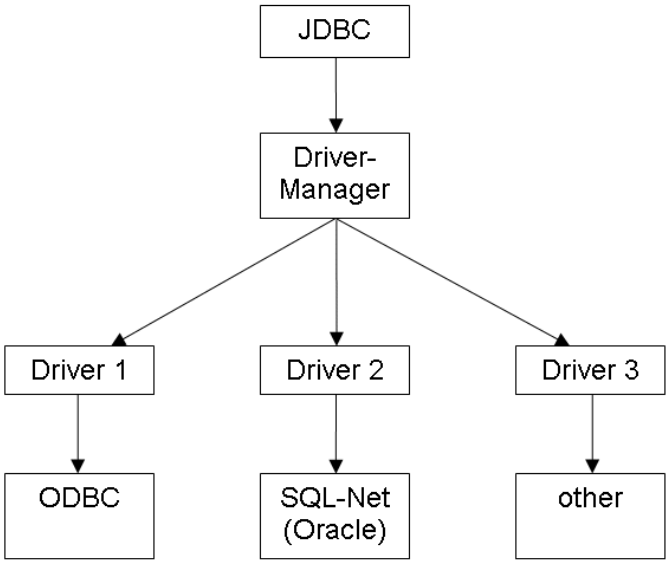


Abbildung 17: Verwaltung von Datenbankverbindungen

JDBC-Treiber lassen sich in vier verschiedene Kategorien unterteilen. Unterschiede bezüglich deren Architektur werden nachfolgend erläutert und anhand von Abbildungen näher veranschaulicht.

3.4.5.1 JDBC-Treiber Typ 1 (JDBC-ODBC-Bridge & ODBC-Driver)

Bei Verwendung des *JDBC*-Treibers Typ 1 wird für den Datenbankzugriff ein nativer *ODBC*-Treiber verwendet, der eine lokale Installation auf dem Client erfordert und über die *JDBC-ODBC*-Bridge von SUN angesteuert wird. Aufgrund der clientseitigen Installation des Treibers ist diese Lösung für den Einsatz im Internet eher ungeeignet. Folgende Abbildung veranschaulicht die Architektur:

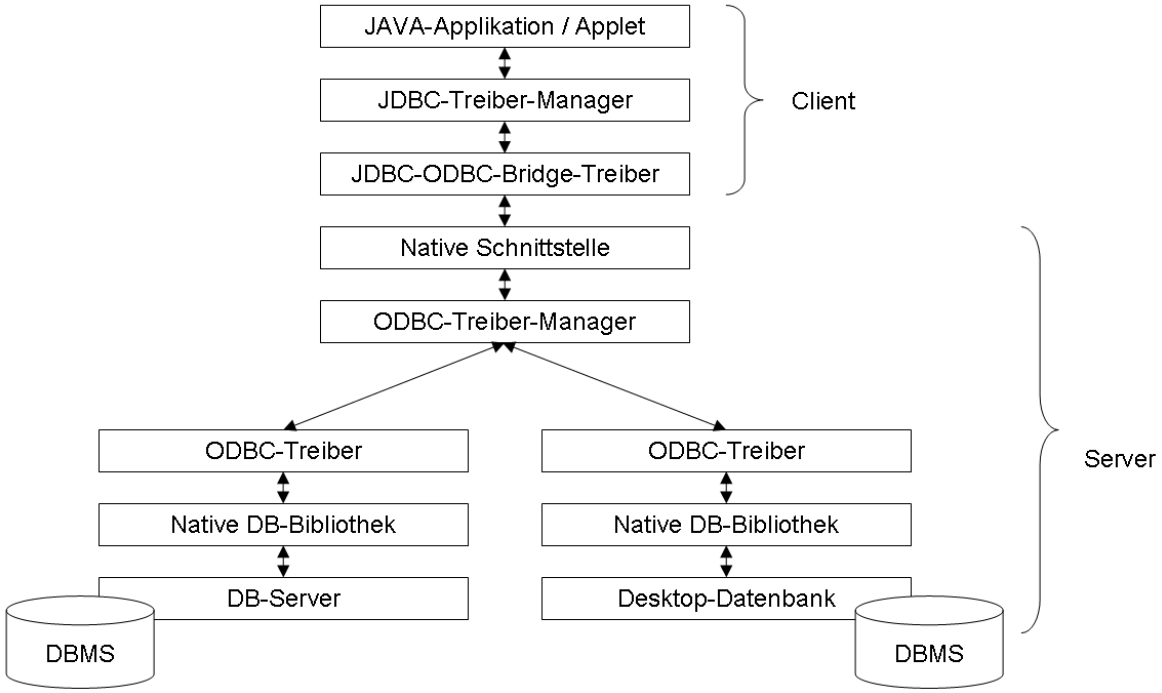


Abbildung 18: JDBC-Treiber Typ 1

Die *JAVA*-Applikation bzw. das *JAVA-Applet* bindet über den *JDBC*-Treiber-Manager den *JDBC-ODBC*-Bridge-Treiber ein, der seinerseits über native Schnittstellen (z.B. *Windows-DLLs*) den *ODBC*-Treiber-Manager in Verbindung mit einem lokal installierten *ODBC*-Treiber zur Datenbankanbindung anspricht.

3.4.5.2 JDBC-Treiber Typ 2 (Native API Partly JAVA Driver)

Bei dieser Architektur wird anstelle des nativen *ODBC*-Treibers ein nativer herstellerabhängiger Treiber (z.B. ein Oracle-Treiber) verwendet. Dieser Treiber muss ebenfalls lokal auf dem Client installiert werden. Die nachfolgende Abbildung veranschaulicht den Aufbau näher:

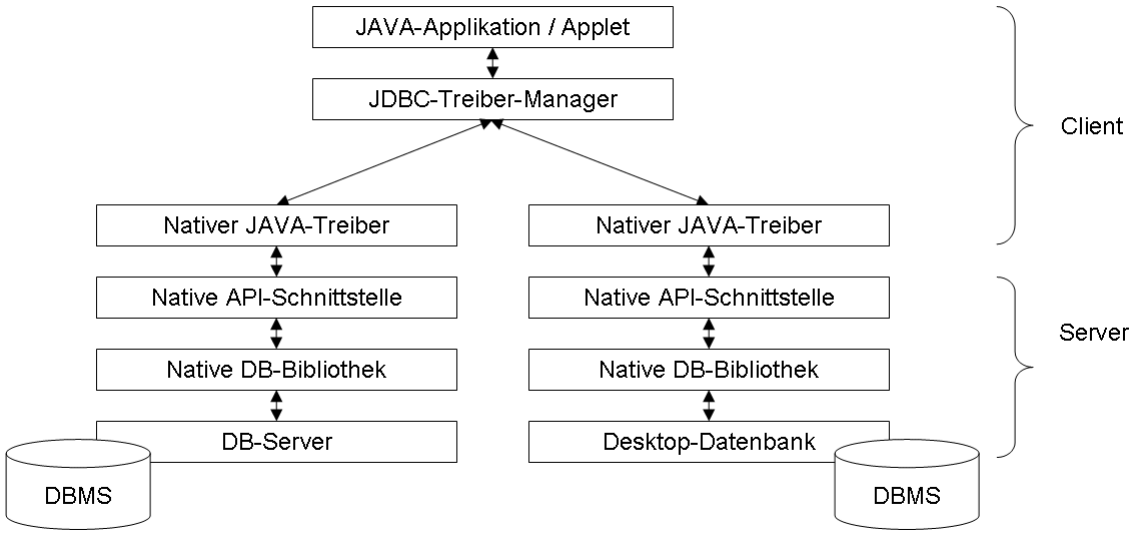


Abbildung 19: JDBC-Treiber Typ 2

Die *JAVA*-Applikation bzw. das *JAVA*-Applet bindet über den *JDBC*-Treiber-Manager einen nativen *JDBC*-Treiber ein, welcher über native *API*-Schnittstellen (z.B. *Windows-DLLs*) die Verbindung zur Datenbank herstellt.

3.4.5.3 JDBC-Treiber Typ 3 (JDBC Net "Pure JAVA" Driver)

Durch diese Architektur wird die lokale Installation eines Treibers auf dem Client durch eine serverseitige Middleware-Installation ersetzt, welche den Datenbankzugriff realisiert. Dieser Treiber-Typ bietet den Vorteil, dass auf der Client-Seite keinerlei Installation mehr notwendig ist, da der universelle Treiber für den Middleware-Zugriff direkt vom Server geladen werden kann und der *DBMS*-spezifische Treiber auf dem Server ausgeführt wird. Die folgende Abbildung veranschaulicht die Funktionsweise dieses Treiber-Typs:

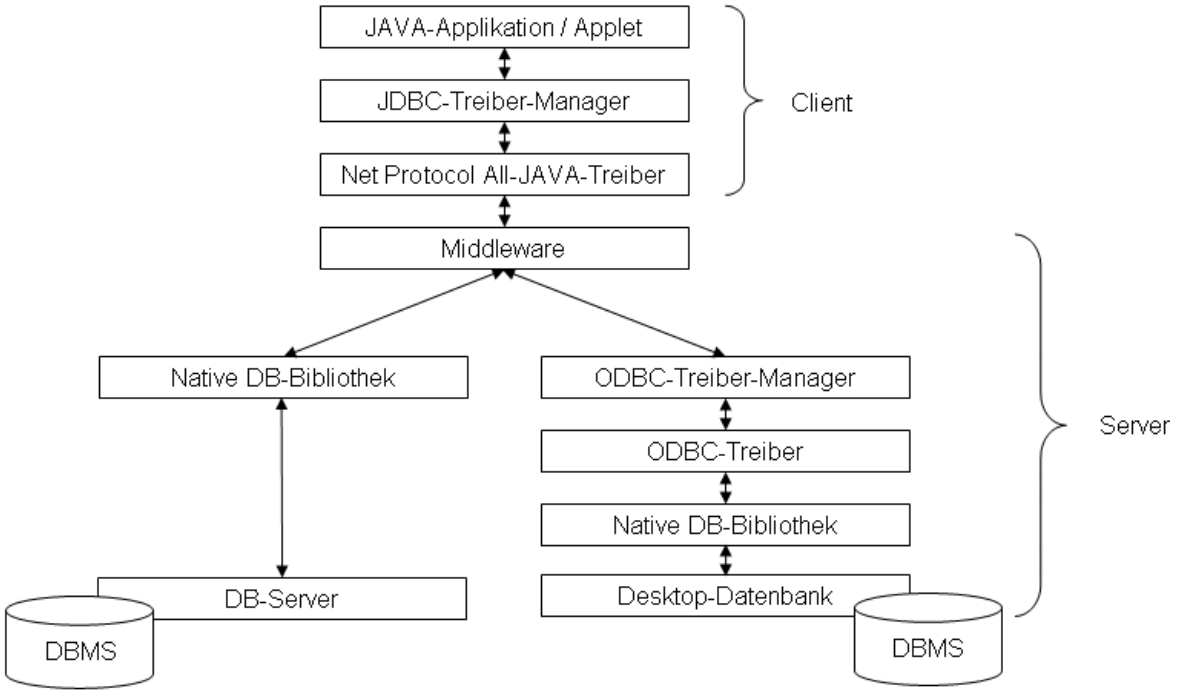


Abbildung 20: JDBC-Treiber Typ 3

Die *JAVA*-Applikation bzw. das *JAVA-Applet* bindet über den *JDBC*-Treiber-Manager einen universellen *JDBC*-Treiber ein, welcher die Aufrufe in ein DBMS-unabhängiges Protokoll übersetzt und anschliessend an die auf dem Server installierte Middleware sendet. Diese wandelt die Aufrufe in datenbankspezifische Aufrufe um und kommuniziert mit der Datenbank.

3.4.5.4 JDBC-Treiber Typ 4 (Native Protocol "Pure JAVA" Driver)

Bei dieser Variante liegt ein Treiber vor, der in purem *JAVA*-Code programmiert ist und somit die Einbindung von nativem Code entfällt. Dieser Treiber-Typ gilt als der Modernste, da durch die fehlende Einbindung von nativem Code sowohl die Plattformunabhängigkeit von *JAVA* unterstützt als auch das Herunterladen des Treibers vom Webserver ermöglicht wird.

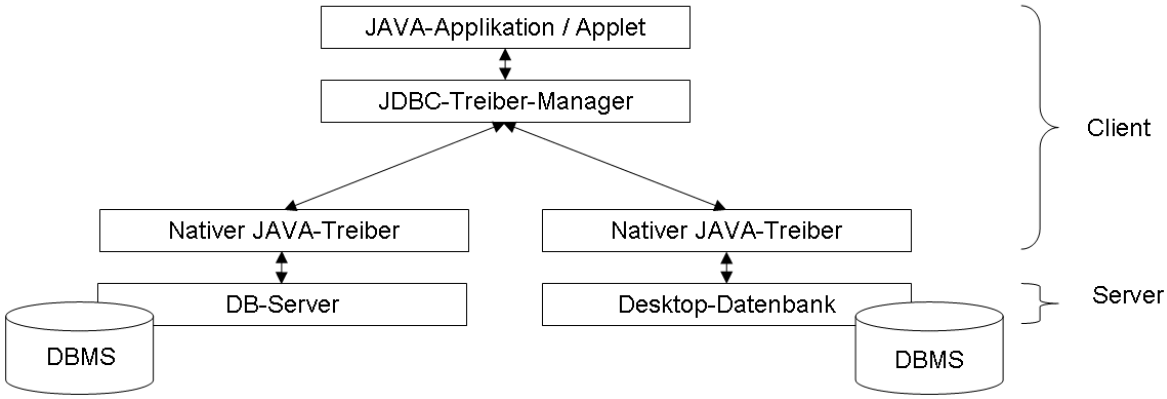


Abbildung 21: JDBC-Treiber Typ 4

Die *JAVA*-Applikation bzw. das *JAVA-Applet* bindet über den *JDBC*-Treiber-Manager einen nativen *DBMS*-spezifischen *JDBC*-Treiber ein, welcher auf direktem Wege die Datenbankverbindung herstellt.

3.4.5.5 Vergleich der unterschiedlichen Treiber-Typen

Bei der Gegenüberstellung der einzelnen Treiber-Typen lässt sich feststellen, dass die Typen 1 und 3 keine Netzwerkfähigkeiten zur Verfügung stellen und ein direkter Zugriff auf ein *DBMS* nicht möglich ist. Diese können somit nur in Verbindung mit *ODBC*-Treibern verwendet werden. Die Treiber-Typen 2 und 4 hingegen besitzen ein Netzwerkprotokoll, wodurch ein direkter Zugriff auf ein *DBMS* ermöglicht wird. Da diese Typen jedoch nicht für alle Datenbanken zur Verfügung stehen, muss oft auf Typ 1 und 3 zurückgegriffen werden.

3.4.6 Grundstruktur einer Anwendung

Generell lässt sich jede *JDBC*-Anwendung bezüglich ihres Ablaufs in drei Bereiche untergliedern. Zuerst wird eine Verbindung zum Datenbankmanagementsystem hergestellt, danach werden Anfragen an die Datenbank formuliert und verschickt. Anschliessend, nach dem Erhalt der Ergebnisse, werden diese verarbeitet [Melton00].

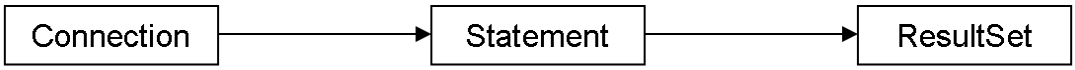


Abbildung 22: Ablauf einer JDBC-Anwendung

Obige Abbildung schildert den Ablauf einer *JDBC*-Anwendung in abstrakter Form. Der exakte Ablauf ist der folgenden Aufzählung zu entnehmen.

- Importieren der notwendigen Klassen und Interfaces.
- Laden des entsprechenden *JDBC*-Treibers.
- Herstellen einer Verbindung zum Datenbankmanagementsystem.
- Erstellen einer Anweisung.
- Ausführen einer Anweisung.
- Abfragen und Verarbeiten der Ergebnisse.
- Beenden der Verbindung zum Datenbankmanagementsystem.

Die in der Aufzählung aufgeführten Schritte werden nun im Folgenden genauer ausgeführt.

3.4.6.1 Importieren der notwendigen Klassen und Interfaces

Alle für die Ausführung von *JDBC* notwendigen Klassen befinden sich im *Package java.sql*, das Bestandteil des *JDK1.4.0* ist. Um diese Klassen verwenden zu können, ist es erforderlich, diese zu Beginn des Programms mit der Anweisung *import java.sql.** zu importieren. Mithilfe des Platzhalters * wird erreicht, dass alle Klassen des angegebenen *Packages* importiert werden.

3.4.6.2 Laden des entsprechenden JDBC-Treibers

Um *JDBC*-Anweisungen ausführen zu können, ist es erforderlich, einen Datenbank-Treiber zu laden, der die Anweisungen in eine Form umsetzt, die vom jeweiligen Datenbankmanagementsystem verstanden wird. Dieser Treiber wird mit dem *JAVA*-Klassenlader über die Methode *class.forName()* geladen.
Die Syntax für das Laden eines *JDBC*-Treibers lautet beispielsweise:
class.forName("oracle.jdbc.driver.OracleDriver");

3.4.6.3 Herstellen einer Verbindung zum Datenbankmanagementsystem

Eine Datenbankverbindung wird über die Methode *getConnection()* hergestellt.
Diese Methode erwartet bis zu drei Übergabeparameter, wobei die beiden letzten Parameter optional sind und nur benötigt werden, wenn die Datenbank eine Authentifizierung verlangt.

- *String IN*: URL der Datenbank, zu der eine Verbindung aufgebaut werden soll.
- *String IN*: Anmeldename für die Datenbankanmeldung.
- *String IN*: Passwort für die Datenbankanmeldung.

Nach einer fehlerfreien Ausführung der Methode wird eine geöffnete Datenbankverbindung vom Typ *Connection* zurückgeliefert.
Die Syntax für den Verbindungsaufbau zu einer Datenbank lautet:
Connection myConnection = DriverManager.getConnection(url, user, password);

3.4.6.4 Erstellen einer Anweisung (Statement)

Um eine Anweisung zu formulieren, ist es erforderlich, eine Instanz einer Anweisung (*Statement*) zu erzeugen. Dies erfolgt mithilfe der Methode *createStatement()* der Klasse *connection.* Hierbei ist zu beachten, dass diese Methode für eine bestehende Datenbankverbindung ausgeführt werden muss, d.h. eine Verbindung zum Datenbankmanagementsystem muss bereits existieren.
Die Syntax für die Erstellung einer Anweisung lautet:
Statement myStatement = myConnection.createStatement();

3.4.6.5 Ausführen einer Anweisung (Statement)

Ein *Statement* wird mittels der Methoden *executeQuery()* für Abfragen oder *executeUpdate()* für Modifikationen der Datenbank ausgeführt. Die beiden Methoden beziehen sich immer auf eine bestehende Anweisung und erhalten als Übergabeparameter jeweils einen String, der die Anfrage in *SQL*-Notation beinhaltet. Wird eine Abfrage abgesetzt, so ist der Rückgabewert vom Typ *ResultSet* und beinhaltet die Ergebnisse in tabellarischer Form.
Bei Modifikationen ist der Rückgabewert vom Typ *int* und enthält die Anzahl der geänderten Datensätze.
Die Syntax für die Ausführung einer Anweisung lautet:

```
ResultSet myResultSet = myStatement.executeQuery( "SELECT * FROM TAB" );
```

bzw.

```
int myResultSet = myStatement.executeUpdate( "UPDATE TAB SET ..." );
```

3.4.6.6 Abfragen und Verarbeiten der Ergebnisse

Der Rückgabewert (*Resultset*) ist ein Objekt, welches die Ergebnisse einer Abfrage oder einer Modifikation enthält. Über Methoden wie *next(), deleteRow(), getArray(), findColumn(), getInt(), getString(),* usw. können die Ergebnisse, die in tabellarischer Form vorliegen, selektiert und bearbeitet werden. Im Allgemeinen entspricht die Form eines *Resultsets* einer Relation mit Spaltenbezeichnung und den dazugehörigen Werten.
Die Syntax für eine Ergebnisabfrage lautet:

```
int nummer = myResultSet.getInt("kundennummer");
```

bzw.

```
String name = myResultSet.getString("name");
```

Mit dem Interface *ResultSet* wird für jeden Datentyp eine entsprechende Methode der Art *getTyp()* angeboten, mittels der die entsprechende Ergebnisspalte ausgelesen werden kann.

3.4.6.7 Beenden der Verbindung zum Datenbankmanagementsystem

Um die Verbindung zu einer Datenbank zu beenden, wird die Methode *close()* der Klasse *connection* bereitgestellt, die für eine vorliegende Verbindung ohne Übergabeparameter aufgerufen wird.
Die Syntax für das Beenden einer Verbindung lautet:
myConnection.close();

3.4.7 Fehlerbehandlung von SQL-Exceptions

JAVA bietet *Exceptions* (Ausnahmen) als spezielle Verfahren zum Umgang und Abfangen von Fehlersituationen. Nahezu jede Methode der *JDBC*-Klassen und -Interfaces kann eine *SQL-Exception* verursachen. Damit der Programmfluss beim Auftreten von *Exceptions* nicht unterbrochen wird, ist es erforderlich, diese zu überwachen und gegebenenfalls besonders ausgezeichnete Programmstücke mit speziellem Code zur Behandlung von *Exceptions* aufzurufen.

3.4.7.1 Exceptions in JAVA mit „try“ und „catch“

Bei dieser Art der Überwachung wird der überwachte Programmbereich durch das Schlüsselwort *try* eingeleitet und durch *catch* beendet. Hinter dem Schlüsselwort *catch* folgt der Programmblock, der beim Auftreten einer *Exception* ausgeführt wird. Man spricht daher bei *JAVA* auch vom „Fangen“ von Fehlern.
Die Syntax für das „Fangen“ von *Exceptions* mittels *try* und *catch* lautet:

```
try {
        String name = myResultSet.getString("name");
}
catch ( SQLException e ) {
        e.printStackTrace();
        return null;
}
```

3.4.7.2 Exceptions in JAVA mit „throws“

Neben der Überwachung von problematischen Programmbereichen durch „*try/catch*“-Blöcke gibt es noch eine andere Möglichkeit, um auf *Exceptions* zu reagieren. Bei dieser wird im Kopf der betreffenden Methode eine *throws*-Klausel eingeführt, die signalisiert, dass die Methode nicht selbst die auftretende *Exception* behandelt, sondern diese an die aufrufende Methode weitergibt. Dies hat zur Folge, dass die aufrufende Methode selbst für die Fehlerbehandlung zuständig ist und der Fehler somit entlang einer Kette von Methodenaufrufen nach oben gereicht wird, wo dieser schliesslich abgefangen und bearbeitet werden muss.

Die Syntax für die Behandlung von *Exceptions* mittels *throws*-Klausel lautet:

```
String readFirstLineFromFile( String filename ) throws FileNotFoundException
{
        RandomAccessFile f = new RandomAccessFile( filename, „r“ );
        return f.readLine();
}
```

3.4.7.3 java.sql.SQLException

Die Klasse *java.sql.SQLException* stellt die Basisklasse aller *JDBC-Exceptions* dar und wird ausgelöst, wenn Datenbankzugriffsfehler auftreten. Um die Möglichkeit zu schaffen, mehrere Fehlerquellen zu behandeln, werden die aufgetretenen Fehler in Form einer verketteten Liste gespeichert und können über von dieser Klasse bereitgestellte Funktionen dargestellt und bearbeitet werden.

Wichtige Methoden dieser Klasse sind:

- *int getErrorCode():*
  Mithilfe dieser Methode wird der herstellerspezifische Fehlercode in Bezug auf die aufgetretene *Exception* zurückgeliefert.
- *SQLException getNextException():*
  Mit dieser Methode wird die nächste Fehlermeldung in der verketteten Liste zurückgegeben bzw. Null, wenn keine weitere Fehlermeldung mehr vorliegt.
- *String getSQLState():*
  Diese Methode ermöglicht es, die aufgetretene *Exception* nach der *XOPEN SQLState*-Konvention anzugeben.

- *void setNextException():*
  Mittels dieser Methode wird ein neues Objekt vom Typ *SQLException* erstellt und der verketteten Liste hinzugefügt.

3.4.7.4 java.sql.SQLWarning

Die Klasse *java.sql.SQLWarning* stellt eine Erweiterung der Klasse *java.sql.SQLException* dar und ermöglicht die Darstellung von Informationen bezüglich Warnungen, die beim Datenbankzugriff aufgetreten sind.
Wie bei der Klasse *java.sql.SQLException* werden die Warnungen in Form einer verketteten Liste gespeichert und lassen sich über entsprechende Funktionen darstellen und bearbeiten.
Wichtige Methoden dieser Klasse sind:

- *SQLWarning getNextWarning():*
  Mit dieser Methode wird die nächste Warnmeldung in der verketteten Liste zurückgegeben bzw. Null, wenn keine weitere Meldung mehr vorliegt.
- *void setNextWarning(SQLWarning w):*
  Mittels dieser Methode wird ein neues Objekt vom Typ *SQLWarning* erstellt und der verketteten Liste hinzugefügt.

Des Weiteren werden die Methoden der Oberklasse *java.sql.SQLException* geerbt.

3.4.7.5 java.sql.DataTruncation

Die Klasse *java.sql.DataTruncation* ist eine Erweiterung der Klasse *java.sql.SQLWarning* und stellt Informationen zur Verfügung, wenn Datenwerte unerwartet abgeschnitten werden. Tritt beispielsweise ein Fehler beim Lesen eines Datenfeldes aus einer Datenbank auf, so dass dieses nur teilweise gelesen wird, so wird eine *SQL-Warning* ausgegeben. Tritt dieser Fehler hingegen beim Schreiben auf, so wird eine *SQL-Exception* ausgelöst.

3.5 JAVA Design Patterns

3.5.1 Einführung

„Ein Entwurfsmuster benennt, motiviert und erläutert systematisch einen allgemeinen Entwurf, der ein in objektorientierten Systemen immer wiederkehrendes Entwurfsproblem löst. Es beschreibt das Muster, die Lösung, wann die Lösung anwendbar ist sowie die Konsequenzen der Anwendung. Es gibt weiterhin Implementierungstips und Beispiele. Die Lösung ist eine allgemeine Anordnung von Objekten und Klassen, die das Problem lösen. Die Lösung wird massgeschneidert und implementiert, um das Problem in einem konkreten Kontext zu lösen.“

(Quelle: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns: Elements of reusable object-oriented Software – Addison-Wesley Publishing Company, Inc., Massachusetts, 1995)

Design Patterns stellen eine der wichtigsten und interessantesten Entwicklungen der objektorientierten Softwareentwicklung der letzten Jahre dar. Sie decken ein bestimmtes Entwurfsproblem ab und beschreiben in rezeptartiger Weise die Anordnung von Objekten, Klassen und Methoden sowie deren Zusammenspiel.
Eine sehr wichtige Rolle standardisierter *Design Patterns* stellt die Namenskonvention dar. Zwar ist es in der Praxis nicht immer möglich, ein bestimmtes *Design Pattern* in allen seinen Details zu übernehmen, die konsistente Verwendung deren Namen und prinzipieller Aufbauweise erweitern jedoch die Kommunikationsfähigkeit des Entwicklers beträchtlich. Begriffe wie *Observer, Factory* oder *Composite* werden in der Entwicklung routinemässig verwendet und haben für Entwickler eine einheitliche Bedeutung [Gamma95].

Die in diesem Kapitel beschriebenen Entwurfsmuster schildern lediglich die verwendeten Muster. Es existieren jedoch eine ganze Reihe weiterer Muster, auf die aufgrund deren Umfangs jedoch nicht näher eingegangen wird.

3.5.2 Observer Pattern

Das Entwurfsmuster *Observer* dient in der Regel der Zustandsüberwachung eines oder mehrerer Objekte. Dabei handelt es sich typischerweise um die Bildschirmdarstellung eines Wertes, wie z.B. ein *Label*, das den aktuellen Kontostand präsentiert. Ändert sich der Kontostand, so soll auch die Benutzeroberfläche den aktuellen Wert anzeigen. Ein solches Problem lässt sich sehr einfach ohne Verwendung eines Entwurfsmusters lösen. Das Kontomodell muss lediglich die *setText()*-Methode des *Labels* aufrufen, um den aktuellen Kontostand auf der Benutzeroberfläche zu aktualisieren. Existieren jedoch neben der eigentlichen Anzeige des Kontostandes noch diverse andere, wie z.B. die Darstellung in Form eines Balken- oder Liniendiagramms, so müssen schon drei Visualisierungen vom Kontomodell verwaltet werden. An dieser Stelle findet das Entwurfsmuster *Observer* seine Anwendung.
Das Entwurfsmuster *Observer* stellt ein *Design Pattern* dar, das eine Beziehung zwischen einem *Subject* und seinen *Observern* aufbaut. Das *Subject* stellt ein Objekt dar, dessen Zustandsänderung für andere Objekte interessant ist. Im obigen Beispiel wird das *Subject* vom eigentlichen Konto repräsentiert. Die Rolle des *Observers* übernimmt die Benutzeroberfläche bzw. das *Label*, das den aktuellen Kontostand anzeigt. Als *Observer* werden all die Objekte bezeichnet, die von Zustandsänderungen des *Subjects* betroffen sind, deren Zustand also dem Zustand des *Subjects* angepasst werden muss.
Generell setzt sich das Entwurfsmuster aus den folgenden Elementen zusammen:

- **Interface *Observer:***
  Das Interface *Observer* definiert eine Methode mit dem Namen *update*, die immer dann aufgerufen wird, sobald sich der Zustand eines *Subjects* geändert hat.
- ***Concrete Observer:***
  Die Klasse *Concrete Observer* stellt den konkreten Beobachter des *Subjects* dar. Sie implementiert das Interface *Observer* und registriert sich bei allen *Subjects*, über deren Zustandsänderung sie unterrichtet werden möchte. Des Weiteren implementiert sie in der Methode *update()* den Code, der bei Zustandsänderungen ausgeführt werden soll.

- ***Subject:***
  In der Klasse *Subject* sind alle Methoden definiert, die für die Registrierung und Deregistrierung der *Observer* notwendig sind. Sobald eine Zustandsänderung des *Subjects* auftritt, wird die Methode *update()* bei allen registrierten *Observern* aufgerufen.
- ***Concrete Subject:***
  Die Klasse *Concrete Subject* ist von der Klasse *Subject* abgeleitet und sorgt für die Speicherung des Zustands. Des Weiteren stellt sie über die Methode *getStatus()* den Dienst zur Abfrage des Zustands bereit. Tritt eine Zustandsänderung des *Subjects* auf, so werden über die Methode *notify()* alle registrierten *Observer* über die Zustandsänderung benachrichtigt.

Das folgende UML-Diagramm (Unified Modeling Language) veranschaulicht die Interaktion zwischen den Klassen bzw. Interfaces:

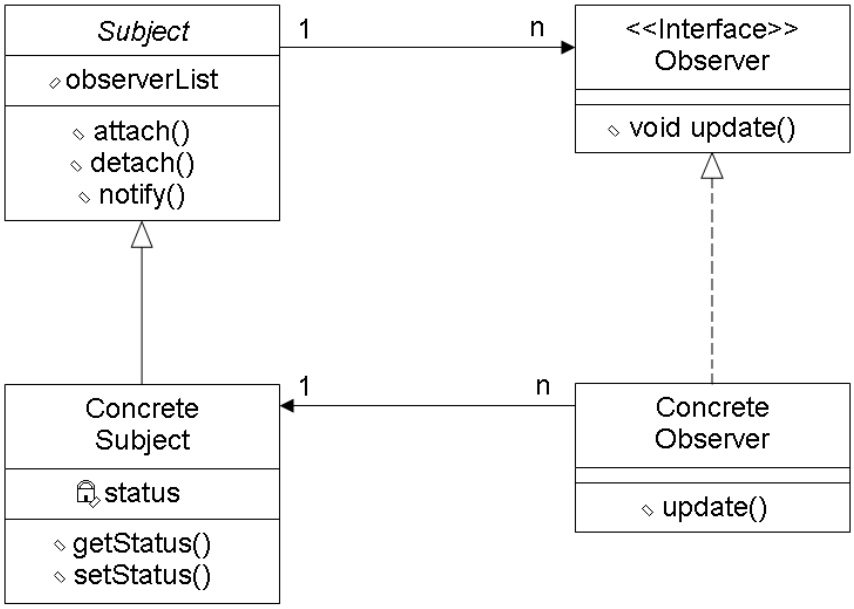


Abbildung 23: UML-Diagramm des Observer Patterns

Das folgende Sequenzdiagramm veranschaulicht die Interaktion zwischen *Concrete Subject* und zweier *Concrete Observer*:

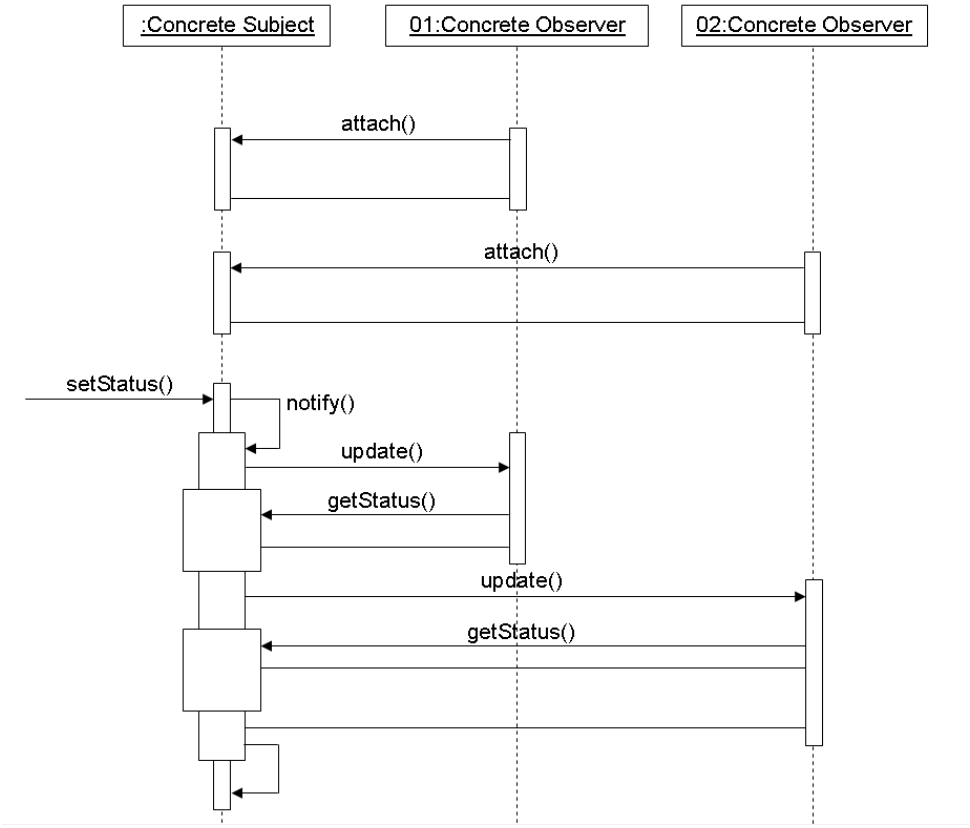


Abbildung 24: Sequenzdiagramm des Observer Patterns

Der oben stehenden Abbildung lässt sich entnehmen, dass sich die beiden *Concrete Observer* über die Methode *attach()* bei dem *Concrete Subject* registrieren, über dessen Zustandsänderung sie unterrichtet werden wollen. Tritt beispielsweise über die Methode *setStatus()* eine Zustandsänderung des *Concrete Subjects* auf, so wird in der abstrakten Klasse *Subject* die Methode *notify()* aufgerufen, welche wiederum über die Methode *update()* alle registrierten *Observer* benachrichtigt und zur Aktualisierung veranlasst. Diese wiederum reagieren auf die Zustandsänderung durch Aufruf der Methode *getStatus()* und aktualisieren ihren zu präsentierenden Zustand.

Das *Observer Pattern* stellt ein in der Anwendungsentwicklung sehr gebräuchliches Entwurfsmuster dar. Die Interaktion zwischen Benutzeroberfläche und Anwendung basiert vollständig auf diesem Konzept [Cooper98].

3.5.3 Abstract Factory Pattern

Um auf das Entwurfsmuster *Abstract Factory* eingehen zu können, ist es zuerst erforderlich, das *Factory Pattern* in kurzer Form zu erklären.
Das *Factory Pattern* stellt generell ein Muster dar, das der Erzeugung von Objekten dient. Es findet dann Einsatz, wenn das Instanzieren eines Objektes mit dem *new*-Operator nicht oder nur sehr schwer möglich ist, da beispielsweise das Objekt sehr schwierig zu konstruieren bzw. zu konfigurieren ist. In diesem Fall ist der Einsatz einer *Factory* sehr sinnvoll, die das Erzeugen neuer Objekte übernimmt.

Das Entwurfsmuster *Abstract Factory* stellt eine Erweiterung des *Factory Patterns* dar und dient der Erzeugung von Objekten, deren Typen in einer bestimmten Beziehung zueinander stehen. Die erzeugten Objekte müssen miteinander verträglich sein und können beispielsweise als Produkte, deren Gesamtheit als Produktgruppe bezeichnet werden. Des Weiteren wird das *Abstract Factory Pattern* auch als *Toolkit* bezeichnet. Ein Beispiel hierfür findet sich in den *JAVA Foundation Classes* bei der Verwendung des *Pluggable Look & Feels*. Die konkrete *Factory* muss hierbei in der Lage sein, diverse Dialogelemente zu erzeugen, die in ihrem Aussehen und ihrer Bedienung ähnlich und konsistent sind, d.h. die Dialogelemente müssen über Fenstergrenzen beständig sein. All diese Dialogelemente bilden zusammen eine Produktgruppe und werden von einer konkreten *Factory* erzeugt. So besteht bei Verwendung von *JAVA Swing* die Möglichkeit, das *Look & Feel* einer Anwendung umzuschalten. Hierbei werden alle Dialogelemente in ihrem Aussehen geändert und die Anwendung erhält ein neues Aussehen. Konkrete *Factories* existieren z.B. für *Macintosh*-, *Windows*- oder X-*Windows*-Oberflächen.

Das folgende UML-Diagramm veranschaulicht die Interaktion zwischen den Klassen:

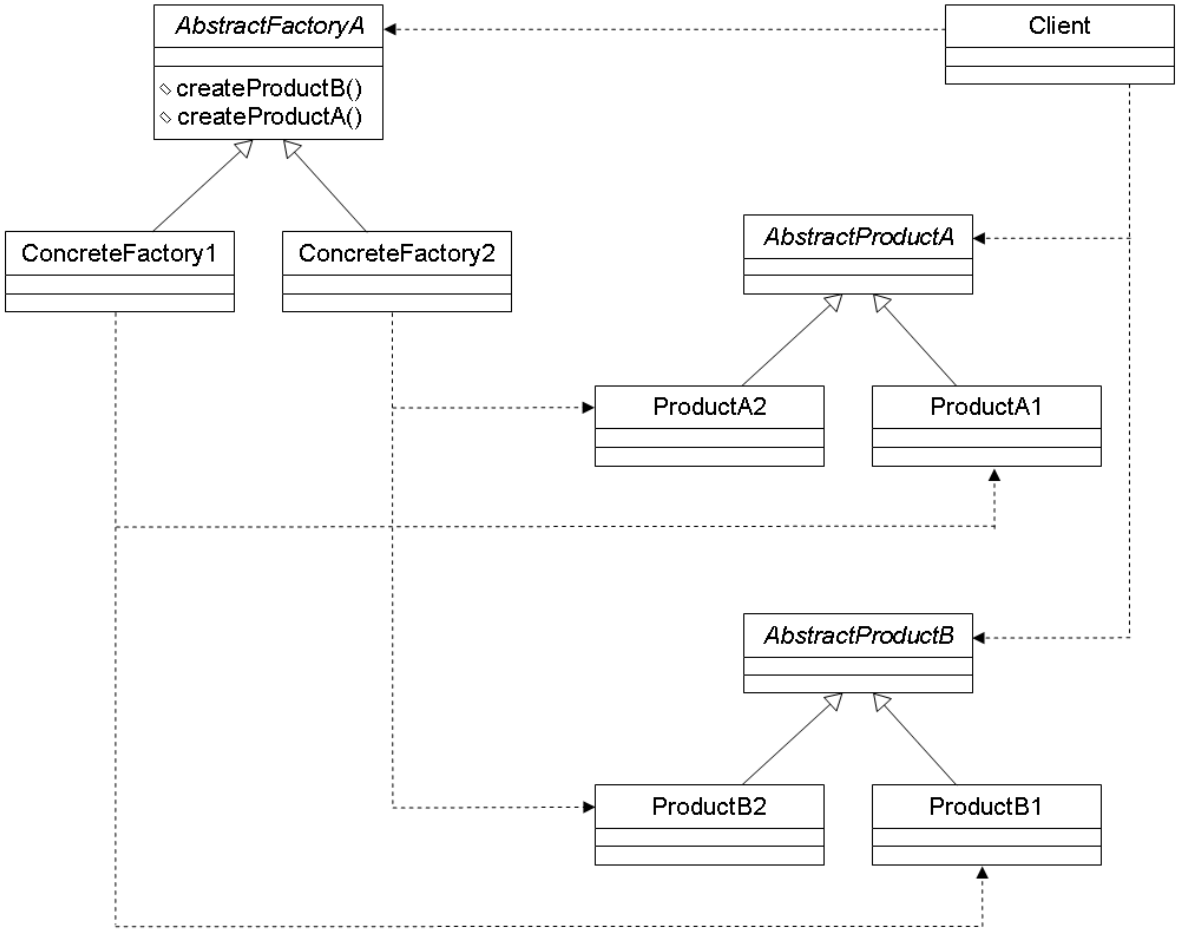


Abbildung 25: UML-Diagramm des Abstract Factory Patterns

Alle *Factory*-Methoden werden in der abstrakten *Factory*-Klasse (*AbstractFactory*) zusammengefasst, die das Interface für die Erzeugung von Produkten darstellt. Für jede Implementierungsvariante existiert eine konkrete *Factory* (*ConcreteFactory*), die von der abstrakten *Factory* abgeleitet wird. Diese enthält alle abstrakten *Factory*-Methoden und ist in der Lage, die konkreten Produkte einer Implementierungsvariante (Produktgruppe) zu erzeugen. Für jede Implementierungsvariante existieren aus den abstrakten Produkten abgeleitete konkrete Produkte, die von der zugehörigen *Factory* erzeugt werden (hier: *ConcreteFactory1* erzeugt *ProductA1* und *ProductB1*, *ConcreteFactory2* erzeugt *ProductA2* und *ProductB2*). Jedes dieser Produkte besitzt eine Basisklasse, die dessen spezifischen Eigenschaften definiert [Cooper98].

3.6 JAVA Beans

3.6.1 Einführung

Eines der Ziele der komponentenbasierten Software-Entwicklung ist es, Anwendungen aus vorgefertigten Bausteinen möglichst einfach und schnell ohne besondere Programmierkenntnisse entwickeln zu können [Flanagan96].
An dieser Stelle setzt das Konzept der *JAVA Beans* an, welche fertige Komponenten für die einfache Erstellung von Anwendungen liefern.
Die Komponenten müssen hierbei die folgenden Eigenschaften aufweisen:

- Sie müssen konfigurierbar sein, wie beispielsweise die Festlegung der Grösse und Farbe einer Schaltflächen-Komponente. Diese Einstellungen müssen persistent sein.
- Sie müssen die Möglichkeit aufweisen, Ereignisse auszulösen. Beispielsweise muss eine Schaltfläche indizieren können, dass sie gedrückt wurde.
- Sie müssen in der Lage sein, auf Ereignisse reagieren zu können.

Um diese Eigenschaften für eine Komponente zu definieren, existieren *Builder-Tools*, die Werkzeuge darstellen, mit denen die Komponenten zu einer lauffähigen Applikation zusammengefügt und entsprechend parametrisiert werden können.

So stellen beispielsweise alle Komponenten des *JAVA-AWT*-Pakets wie auch des *JAVA-Swing*-Pakets *Beans* dar.

Nach einer Definition von SUN sind die wichtigsten Merkmale einer *Bean*:

- **Properties:** Sie stellen die konfigurierbaren Eigenschaften einer *Bean* dar.
- **Methoden:** *JAVA*-Methoden, die von anderen Komponenten oder aus der Umgebung aufgerufen werden können.
- **Events:** Sie stellen eine Möglichkeit der Verbindung von *Beans* zur Verfügung, so dass diese untereinander kommunizieren können.

Im Folgenden wird auf die *Properties* und *Events* einer *Bean* näher eingegangen, da diese in der entwickelten Applikation bei der Verwendung des *Observer Patterns* ihren Einsatz finden.

3.6.2 Properties

3.6.2.1 Einfache Properties

Einfache *Properties* stellen die konfigurierbaren Eigenschaften einer *Bean* dar und verfügen über diverse *Set*- und *Get*-Methoden mit denen die Eigenschaften gesetzt oder gelesen werden können.
Diese haben die folgende Syntax:

- *void set<Eigenschaftsname> (Eigenschaftstyp e)*
- *Eigenschaftstyp get<Eigenschaftsname> ()*

3.6.2.2 Bound Properties

Bound Properties bezeichnen *Properties*, die mittels *Events* andere *Beans* darüber informieren können, dass sich ihr Zustand geändert hat. Bei der Verwendung dieses Konzeptes melden sich interessierte *Listener* bei dem Objekt an, welches die Ereignisse „feuert". Die Verwaltung der *Listener* wird durch die Klasse *java.beans.PropertyChangeSupport* unterstützt. Mit der Erzeugung eines Objektes vom Typ *PropertyChangeSupport* ist es möglich, über die Methoden *addPropertyChangeListener() und removePropertyChangeListener()* das Management der *Listener* zu übernehmen. Hierbei dient die Methode *addPropertyChangeListener()* der Registrierung, die Methode *removePropertyChangeListener()* der Deregistrierung eines *Listeners*.
Findet eine Zustandsänderung einer *Bean* statt, so ist diese in der Lage, mittels Aufruf der Methode *firePropertyChange(PropertyName, alterWert, neuerWert)* die registrierten *Listener* hierüber zu informieren. Diese wiederum können das *PropertyChangeListener*-Interface implementieren und die Methode *propertyChange()* bereitstellen, welche bei einer Zustandsänderung aufgerufen wird.
Des Weiteren werden über die Klasse *PropertyChangeEvent* die Methoden *getOldValue()* und *getNewValue()* zur Verfügung gestellt, die es möglich machen, Zustandsinformation zu erhalten.

3.6.2.3 Constraint Properties

Diese Variante, die hier nur kurz angesprochen wird, stellt eine Ergänzung des Konzeptes der *Bound Properties* dar. *Constraint Properties* ermöglichen es, registrierten *Listenern* ein Veto gegen Änderungen einzulegen. Liegt eine bevorstehende Zustandsänderung einer *Bean* vor, so werden die registrierten *Listener* darüber informiert und die Änderung der *Property* erst durchgeführt, wenn keiner der *Listener* ein Veto dagegen einlegt.

3.6.3 Events

Events spielen im Konzept der *JAVA Beans* eine sehr wichtige Rolle. Um die Möglichkeit einer Verbindung bzw. Kommunikation zwischen *Beans* zu schaffen, ist es erforderlich, das dem *Observer Pattern* ähnelnde *Design Pattern* einzuhalten.

Die Ereignisquellen-*Bean* generiert spezifische Ereignisse, welche durch ein *Listener*-Interface spezifiziert werden und definiert die Methode in diesem, welche bei der Ereignisauslösung aufgerufen wird. Das folgende Beispiel veranschaulicht den Aufbau eines solchen *Listener*-Interfaces:

```
public interface ModelChangedListener extends EventListener {
        public void modelChanged( ModelChangedEvent e );      // Methode, die bei
                                                              Ereignisauslösung
                                                              aufgerufen wird
}
```

Des Weiteren stellt das Ereignisquellen-*Bean* zwei Methoden für die Registrierung bzw. Deregistrierung von *Listenern* zur Verfügung.

```
public void addModelChangedListener( ModelChangedListener m )
public void removeModelChangedListener( ModelChangedListener m )
```

Die Ereignisauslösung erfolgt über die interne private Methode
fireModelChanged(ModelChangedEvent e),
die für jeden registrierten *Listener* die Methode
modelChanged(ModelChangedEvent e) aufruft.

Das Ereignisverarbeitungs-*Bean* hingegen muss das *Listener*-Interface implementieren. Dies erfolgt folgendermassen:

```
class ClassName implements ModelChangedListener {
        .....
}
```

Des Weiteren muss die *Bean* die Methode, die bei Ereignisauslösung aufgerufen wird, implementieren, um konkret auf dieses Ereignis reagieren zu können.

```
public void modelChanged( ModelChangedEvent e ) {
        .....
}
```

Die Registrierung der Ereignisverarbeitungs-*Bean* bei der Ereignisquellen-*Bean* kann entweder durch diese selbst im Konstruktor oder durch ein anderes Objekt erfolgen.

4 Systementwurf

In diesem Kapitel wird der Aufbau des Systems in seinen wesentlichen Elementen sowie dessen Architektur beschrieben und anhand diverser Screenshots ein erster Eindruck des Systems vermittelt.

4.1 Konzeption der Systemarchitektur

4.1.1 Allgemeine Konzeption und Aufbau des Systems

Aufgrund der bereits in Kapitel 1.3 erwähnten hierarchischen Anordnung der zu verwaltenden Facilities wird das gesamte System in verschiedene Ebenen untergliedert und für die jeweiligen Facilities gesonderte Benutzeroberflächen bereitgestellt. So existiert je eine eigene Benutzeroberfläche für Switches, Cabinets, Shelves und Cards. Der Wechsel zwischen den Ebenen erfolgt über bereitgestellte Funktionen, die über einen Doppel-Click mit der linken Maustaste auf das selektierte Listenelement bzw. über eine Schaltfläche in der *MenuBar* oder *ToolBar* aufgerufen werden. Dieser Wechsel ist in der folgenden Abbildung veranschaulicht.

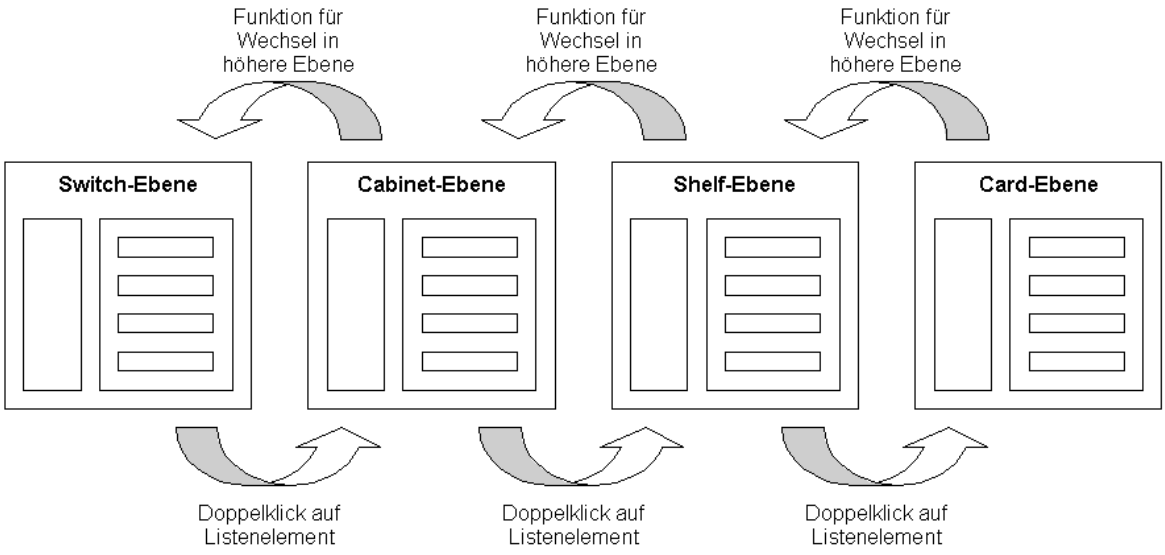


Abbildung 26: Wechsel zwischen den Ebenen

Mit dem Start der Applikation wird eine Verbindung zur Datenbank geschaffen. Ist diese hergestellt, so müssen grundlegende Einstellungen, wie beispielsweise die Festlegung der Sprache und das an die Plattform angelehnte *Look & Feel,* getroffen werden. All diese Vorkehrungen werden in der Klasse *Main* getroffen. An dieser Stelle ist es nun erforderlich, die Rahmenstruktur der Benutzeroberfläche zu erstellen. Dies erfolgt über die Erzeugung einer Instanz der Klasse *Gui*, welche das Grundgerüst der Benutzeroberfläche mit ihrer *MenuBar* und *ToolBar* erzeugt.
Die folgende Abbildung veranschaulicht dies näher:

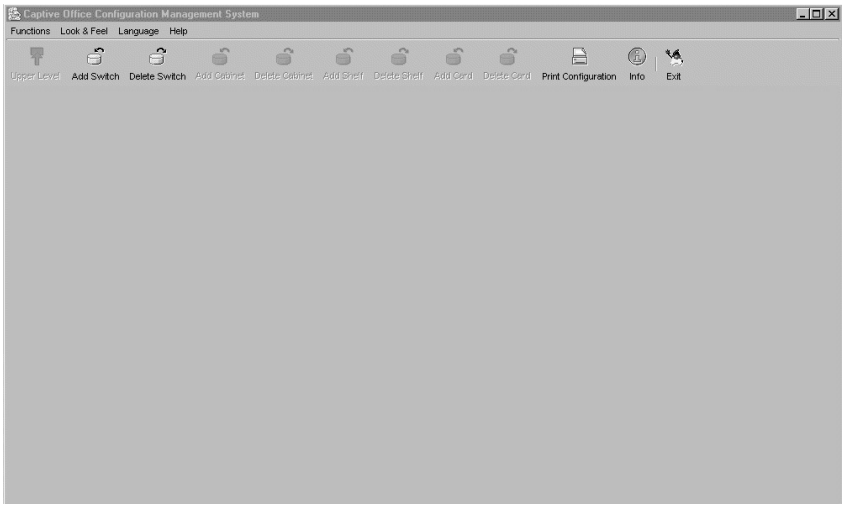


Abbildung 27: Rahmenstruktur der Applikation

Des Weiteren werden in dieser Klasse ebenfalls die Panels erzeugt, welche die für die jeweilige Ebene relevanten Daten involvieren. Diese werden jedoch zu Beginn in den unsichtbaren Zustand gesetzt mit Ausnahme des Panels der obersten Ebene. Dieses Panel wird in den sichtbaren Zustand gebracht.
An dieser Stelle angelangt, befindet sich die Applikation in ihrem Ausgangszustand.

Die folgende Abbildung zeigt die Anordnung der Panels:

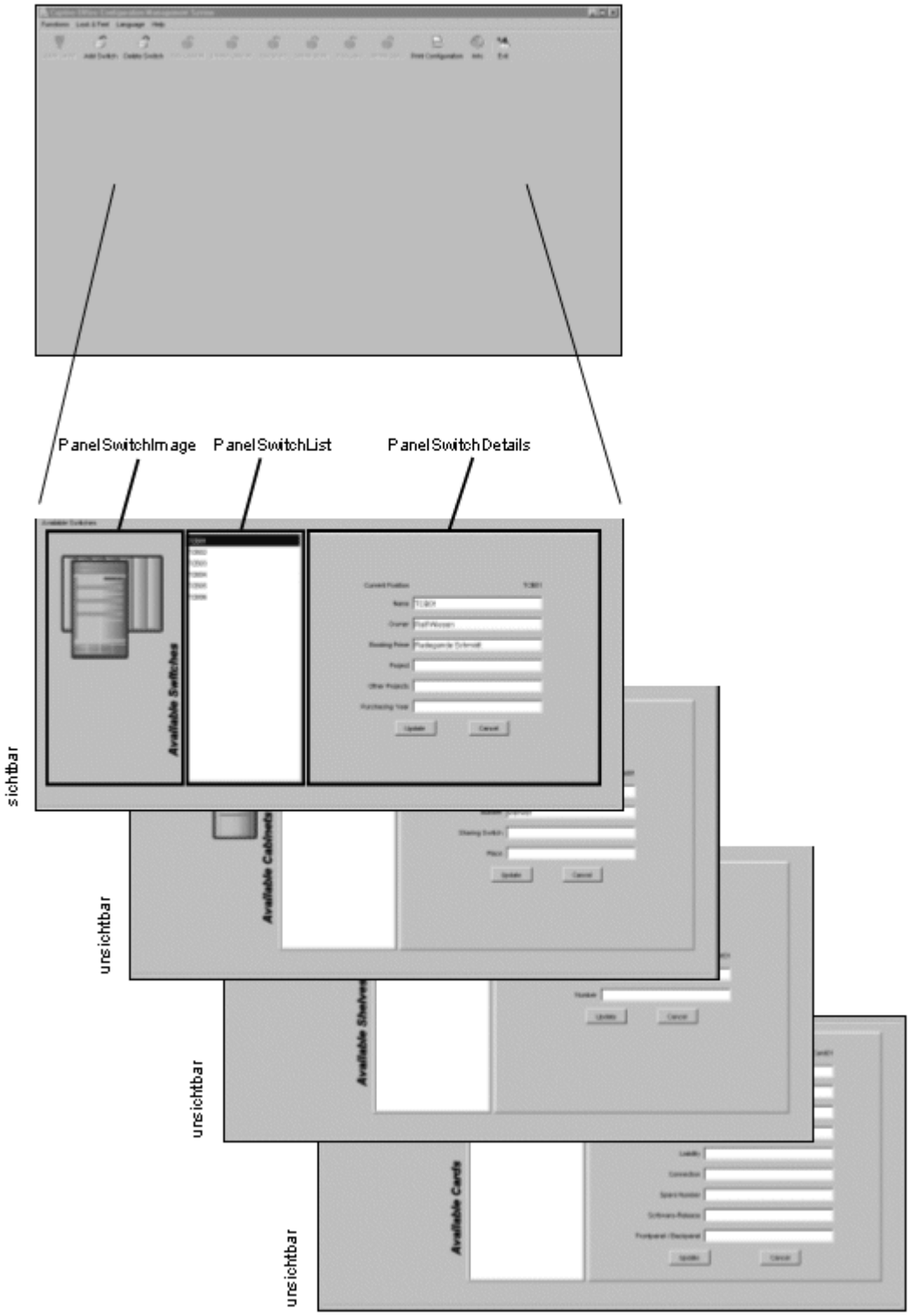


Abbildung 28: Anordnung der Panels

Jedes der erzeugten Panels instanziert bei seiner Erzeugung drei weitere Sub-Panels, die das Symbol der jeweiligen Ebene, die Liste mit den der Ebene zugeordneten Facilities sowie die Attribute des selektierten Listenelements darstellen. Beispielsweise werden aus dem Panel *PanelSwitch* drei Sub-Panels mit den Namen *PanelSwitchImage*, *PanelSwitchList* und *PanelSwitchDetails* erzeugt. Jedes dieser Panels wird durch eine eigene Klasse vertreten.
Die folgende Abbildung zeigt dies in einer baumförmigen Struktur:

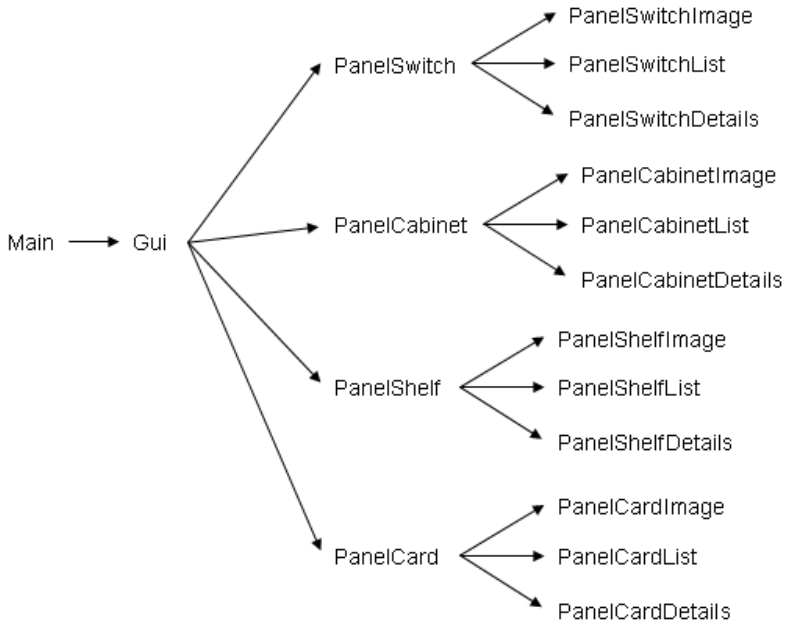


Abbildung 29: Erzeugung der Panels

Um die für die jeweilige Ebene spezifischen Daten präsentieren zu können, ist es erforderlich, diese aus der Datenbank auszulesen. Dies erfolgt jedoch nicht aus den obenstehenden Klassen heraus, da diese lediglich für die Präsentation der Daten zuständig sind, aber nicht für den Datenbankzugriff. Hierfür muss eine gesonderte Klasse existieren, die den Datenbankzugriff realisiert. An dieser Stelle setzt auch das Konzept der Schichtenarchitektur einer Anwendung an, die eine Trennung der Darstellungsebene von der Anwendungslogik nahelegt. Aufgrund der Abhängigkeit der Panels *PanelXXXList* und *PanelXXXDetails* von den Daten, liegt es nahe, an dieser Stelle das Entwurfsmuster *Observer* einzusetzen. Dieses trennt deutlich die Darstellungsebene von den Daten und ermöglicht es, die Benutzeroberfläche in Abhängigkeit von den Daten zu aktualisieren.

4.1.2 Einsatz von Entwurfsmustern und JAVA Beans

Für den Zugriff auf die Datenbank existiert eine Klasse *ResultSetListModel,* die in ihrem Konstruktor in Abhängigkeit vom Übergabeparameter die jeweilige Relation aus der Datenbank liest und somit den jeweiligen Ergebnisdatensatz repräsentiert. Über diverse Methoden stellt diese Klasse eine Art Schnittstelle zur Datenbank zur Verfügung, die es erlaubt, Datensätze zu lesen, der Datenbank Elemente hinzuzufügen, zu modifizieren und zu entfernen.
In Bezug auf das Entwurfsmuster *Observer* stellt diese Klasse das *Concrete Subject* dar, wobei die Datenbank der zu überwachende Zustand ist. Die Benutzeroberflächen, insbesondere die Panels *PanelXXXDetails*, welche die Attribute präsentieren, fungieren als *Concrete Observer* und implementieren das Interface *Observer*. Tritt eine Zustandsänderung des *Concrete Subjects* auf, so werden alle registrierten *Observer* über die Zustandsänderung benachrichtigt und aktualisieren sich entsprechend.
An dieser Stelle empfiehlt sich der Einsatz von *Bound Properties*, wie sie bei den *JAVA Beans* vorliegen. Mittels diesen wird eine einfache Möglichkeit der Realisierung des *Observer Patterns* geboten. Die Benutzeroberflächen (*Concrete Observer*) melden sich als interessierte *Listener* mittels der durch die *JAVA Beans* bereitgestellten Methode *addPropertyChangeListener()* bei dem Objekt (*ResultSetListModel*) an, welches bei Zustandsänderung die Ereignisse „feuert". Diese Ereignisauslösung erfolgt mittels der Methode *firePropertyChange()*, welche für jeden registrierten *Listener* die Methode *modelChanged()* aufruft. Die Methode *modelChanged()* ist in jedem registrierten *Listener* implementiert und führt alle zur Aktualisierung der Benutzeroberfläche notwendigen Befehle durch. Wird beispielsweise ein Element der Datenbank hinzugefügt, so muss dieser Datenbankzugriff über die Klasse *ResultSetListModel* erfolgen. Diese Klasse „feuert" daraufhin in der entsprechenden Methode für das Hinzufügen eines Elements ein Ereignis, welches die angemeldeten *Listener* dazu veranlasst, ihre Methode *modelChanged()* zur Aktualisierung der Benutzeroberfläche aufzurufen. Um eine Aktualität der Benutzeroberfläche sicherstellen zu können, muss jegliche Modifikation von Daten in der Datenbank eine Ereignisauslösung über die Methode *firePropertyChange()* zur Folge haben.

Die folgende Abbildung zeigt beispielhaft das Sequenzdiagramm und schildert den prinzipiellen Ablauf zwischen *Concrete Subject* und *Concrete Observer*:

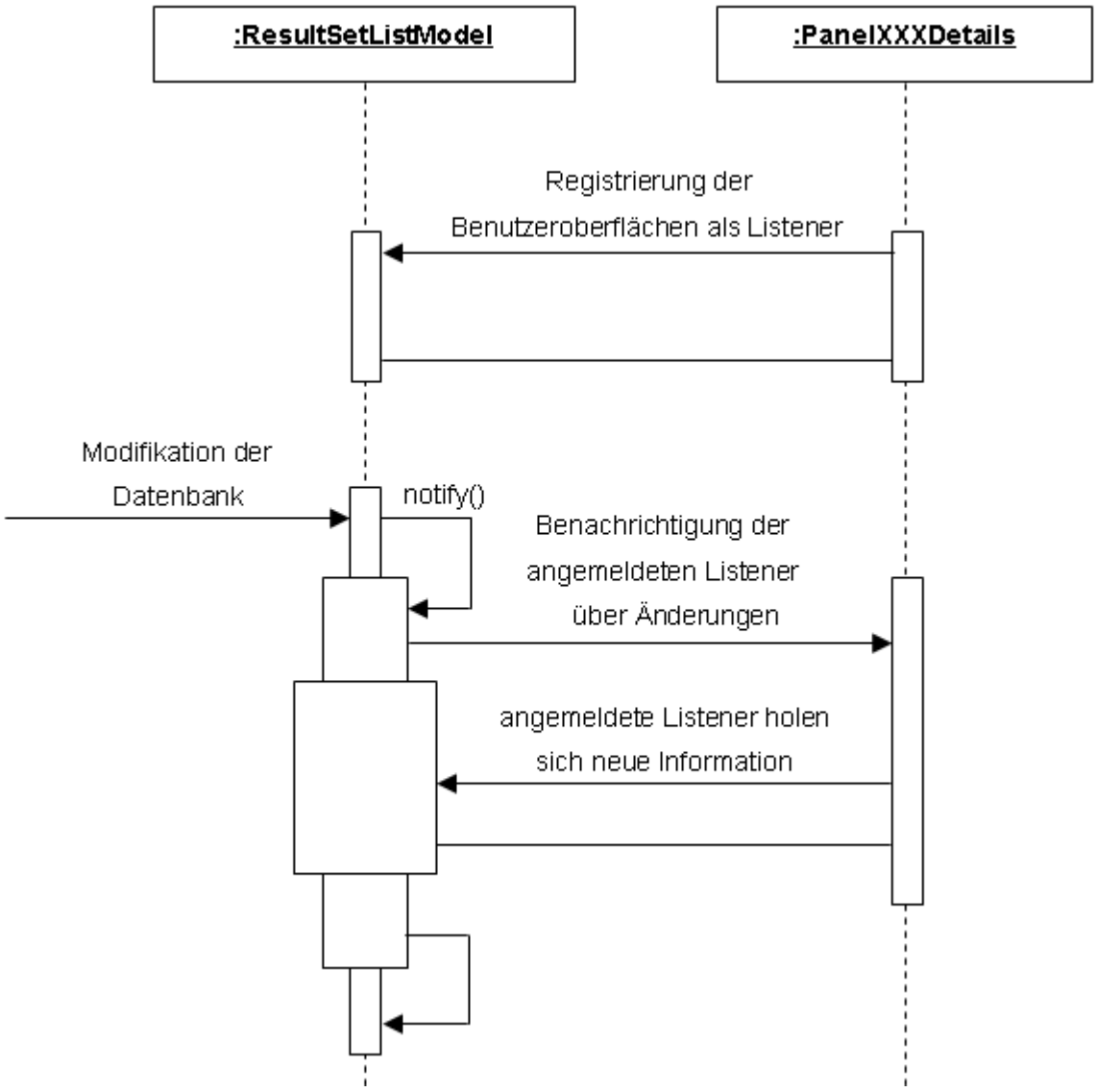


Abbildung 30: Sequenzdiagramm des Observer Patterns im System

Neben dem Entwurfsmuster *Observer* kommt *das Abstract Factory Pattern* zum Einsatz, welches die Grundlage für ein veränderbares *Look & Feel* darstellt. Die *Abstract Factory* ist im wesentlichen ein *Factory*-Objekt, das eines der zur Verfügung stehenden *Factory*-Objekte erzeugt. In *Swing* entspricht die Klasse *LookAndFeel* dieser *Abstract Factory*, die *Factory*-Objekte vom Typ *ComponentUI* erzeugt und verwaltet. Der *UIManager* liefert standardmässig ein *LookAndFeel*-Objekt mit den zur aktuellen Plattform passenden Beschreibungen für die Elemente der Benutzeroberfläche. D.h. die Elemente der Benutzeroberfläche werden in ihrem Aussehen an die jeweils vorliegende Plattform angelehnt. Die Erzeugung der Klasse *LookAndFeel* erfolgt über den Aufruf *Class.forName().* Anschliessend wird diese dem *UIManager* zugewiesen. Über die Methode *getDefaults()* wird auf eine Tabelle referenziert, die Informationen bezüglich des jeweils eingestellten *Look & Feels* für die einzelnen Elemente der Benutzeroberfläche beinhaltet.
Untenstehende Abbildung veranschaulicht den Aufbau in UML-Notation:

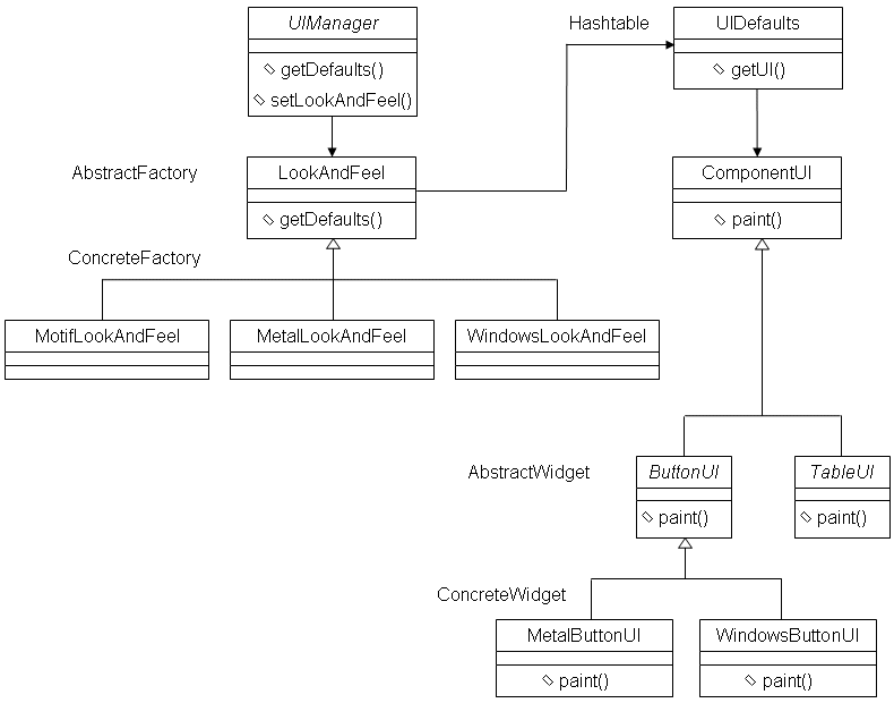


Abbildung 31: UML-Diagramm des Abstract Factory Patterns unter Verwendung von JAVA Swing

Nun ist es möglich, unabhängig von der aktuell benutzen Plattform das *Look & Feel* zu setzen. Hierfür muss lediglich dem *UIManager* mittels der Methode *setLookAndFeel()* ein neues *Look & Feel*-Objekt übergeben werden.

4.1.3 Einsatz der Internationalisierung

Unter dem Begriff der Internationalisierung versteht man den Prozess der Erstellung einer Anwendung, die auch von anderen Kulturen und Sprachen genutzt werden kann. Für die grösste Anzahl von Programmen erweist sich die Übersetzung dieser in eine andere Sprache als sehr schwierig. Zeichenketten für Schaltflächenbeschriftungen werden häufig in das Programm fest compiliert und bedürfen der manuellen Übersetzung durch den Programmierer für das jeweilige Bestimmungsland. Der Vorgang dieser Konvertierung wird auch als Lokalisierung bezeichnet und stellt viel Arbeit für den Übersetzer dar. Des Weiteren resultieren hieraus verschiedene Versionen desselben Programms für unterschiedliche Kulturen und Sprachen. Aufgrund dieser Umstände ist es empfehlenswert, die kultur- und sprachspezifischen Teile von den anderen Teilen eines Programmes zu trennen.
Man spricht hierbei von der sogenannten Internationalisierung, die die Erstellung internationaler Anwendungen wesentlich erleichtert und in dieser Arbeit ihren Einsatz findet.

4.1.3.1 Verwendung von Locales

Eine *Locale* ist ein Objekt, das neben einer Sprache auch eine Kultur spezifiziert. Dies ist notwendig, da sich beispielsweise britisches Englisch einer anderen Formatierung von Zahlen und Datumsangaben bedient, als amerikanisches Englisch. *Locales* werden in *JAVA* durch die Klasse *Locale* definiert, welche einen Feldnamen für eine spezifische *Locale* einsetzt. Die Bezeichnung verschiedener *Locales* ist standardisiert. In dieser Arbeit werden *Locales* für die Sprachen Englisch (US), Deutsch und Französisch verwendet, welche sich der folgenden Tabelle entnehmen lassen:

Locale	**Sprache**	**Land**
en_US	Englisch	USA
DE_DE	Deutsch	Deutschland
fr_FR	Französisch	Frankreich

Tabelle 5: Internationale Sprach- und Ländercodes

4.1.3.2 Verwendung von Resource Bundles

Resource Bundles stellen eine Art der Strukturierung von Zeichenketten dar, wobei verschiedene Zeichenketten für unterschiedliche *Locales* verwendet werden. Eine Schaltfläche kann beispielsweise eine Beschriftung in englisch, deutsch oder französisch besitzen. So werden die Beschriftungen nicht im Quelltext angegeben, sondern über *Resource Bundles* in Textdateien ausgelagert.
Die folgende Abbildung zeigt einen Ausschnitt einer Textdatei für die deutsche Sprache:

```
update = Aktualisieren
duplicate = Duplizieren
owner = Eigentümer
bookingPrime = Booking Prime
project = Projekt
otherProjects = Weitere Projekte
purchasingYear = Anschaffungsjahr
functions = Funktionen
upperLevel = Höhere Ebene
cloneWindow = Fenster dupl.
addSwitch = Switch hinzuf.
deleteSwitch = Switch entf.
addCabinet = Cabinet hinzuf.
deleteCabinet = Cabinet entf.
addShelf = Shelf hinzuf.
deleteShelf = Shelf entf.
addCard = Karte hinzuf.
deleteCard = Karte entf.
printConfiguration = Konfig. drucken
exit = Beenden
```

Tabelle 6: Ausschnitt einer Textdatei für die deutsche Sprache

Resource Bundles werden mit dem Standard-Klassenladeprogramm in eine Anwendung geladen. Deren Dateinamen leiten sich aus einem *Bundle*-Namen her, gefolgt von dem Namen der verwendeten *Locale*. Bei Verwendung der englischen Sprache wird beispielsweise in der entwickelten Anwendung das *Resource Bundle* mit dem Namen *MessagesBundle_en_US.properties* geladen. Existiert dieses nicht, so wird auf ein *Resource Bundle* mit den Standard-Zeichenketten zurückgegriffen, welches ebenfalls verwendet wird, wenn keine Spracheinstellung vorgenommen wird. Dieses trägt in der Anwendung den Namen *MessagesBundle.properties*.

4.1.3.3 Zugriff auf Resource Bundles

Der Zugriff auf die in den *Bundles* gespeicherten Ressourcen erfolgt über Methoden wie beispielsweise *getString()*, *getObject()*, etc., die durch die Klasse *ResourceBundle* zur Verfügung gestellt werden.
Die Beschriftung einer Schaltfläche mit den in den Textdateien gespeicherten Zeichenketten erfolgt beispielsweise folgendermassen:

Button ok = new Button(messages.getString("ok"));

4.1.3.4 Wechsel von Resource Bundles

Über die Klasse *ResourceBundle* wird die Methode *getBundle()* zur Verfügung gestellt, welche es ermöglicht, mittels Übergabe des *Bundle*-Namens und der *Locale* ein neues *Resource Bundle* zu laden. Demnach ist für den Wechsel der Sprach- und Kultureinstellungen einer Anwendung lediglich die neue Festlegung einer *Locale* erforderlich. Dies erfolgt in der Anwendung mittels dem in der *MenuBar* hierfür vorgesehenen Menü.

Tabelle 7: Menü zur Einstellung der Sprache

4.2 Konzeption und Entwurf der Benutzeroberfläche

Ist die Verbindung zur Datenbank hergestellt, so werden die Benutzeroberfläche der Applikation aufgebaut und die für die dargestellte Ebene relevanten Daten aus der Datenbank ausgelesen. Der Anwender befindet sich nun auf der obersten Ebene. Auf dieser werden alle verfügbaren Switches mit ihren zugehörigen Attributen präsentiert. Die folgende Abbildung zeigt den Aufbau der Switch-Ebene:

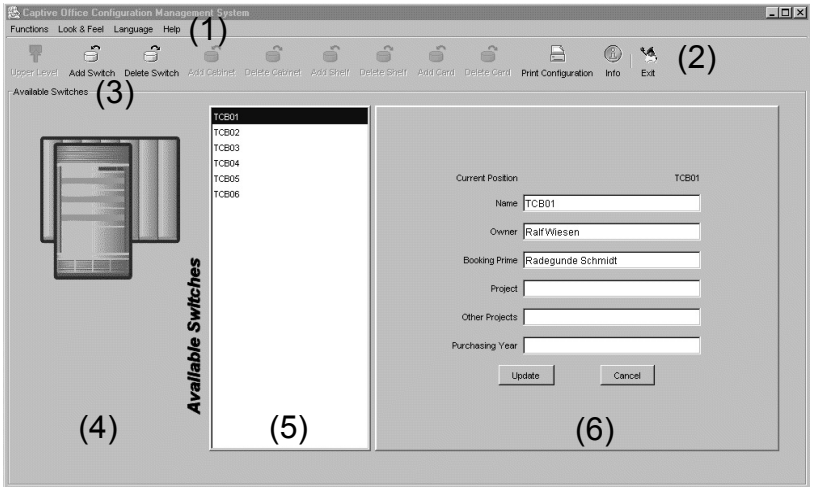


Abbildung 32: Screenshot der Switch-Ebene

Der obigen Abbildung lässt sich entnehmen, dass sich die Benutzeroberfläche generell aus 6 Teilbereichen zusammensetzt.
Diese sind

- (1) Die *MenuBar*, welche die Pull-Down-Menüs *Functions, Look & Feel, Language* und *Help* beinhaltet.
- (2) Die *ToolBar*, welche dieselben Funktionen wie die Pull-Down-Menüs der *MenuBar* beinhaltet, jedoch einen schnelleren Zugriff auf diese ermöglicht.
- (3) Das Panel AvailableSwitches, das wiederum die drei Sub-Panels *PanelSwitchImage, PanelSwitchList* und *PanelSwitchDetails* involviert.
- (4) Das Sub-Panel *PanelSwitchImage*, das die Ebene mittels einer Grafik präsentiert.
- (5) Das Sub-Panel *PanelSwitchList*, das innerhalb einer Liste die Namen der in der Datenbank gefundenen Switches beinhaltet.
- (6) Das Sub-Panel *PanelSwitchDetails*, das die Attribute des in der Liste selektierten Switches präsentiert.

Im Gegensatz zu den Panels stellen die Benutzeroberflächen-Elemente *MenuBar* und *ToolBar* statische Elemente dar, die während der gesamten Nutzungsdauer der Applikation sichtbar sind. Sie bilden quasi die Rahmenstruktur der Anwendung und können der folgenden Abbildung entnommen werden:

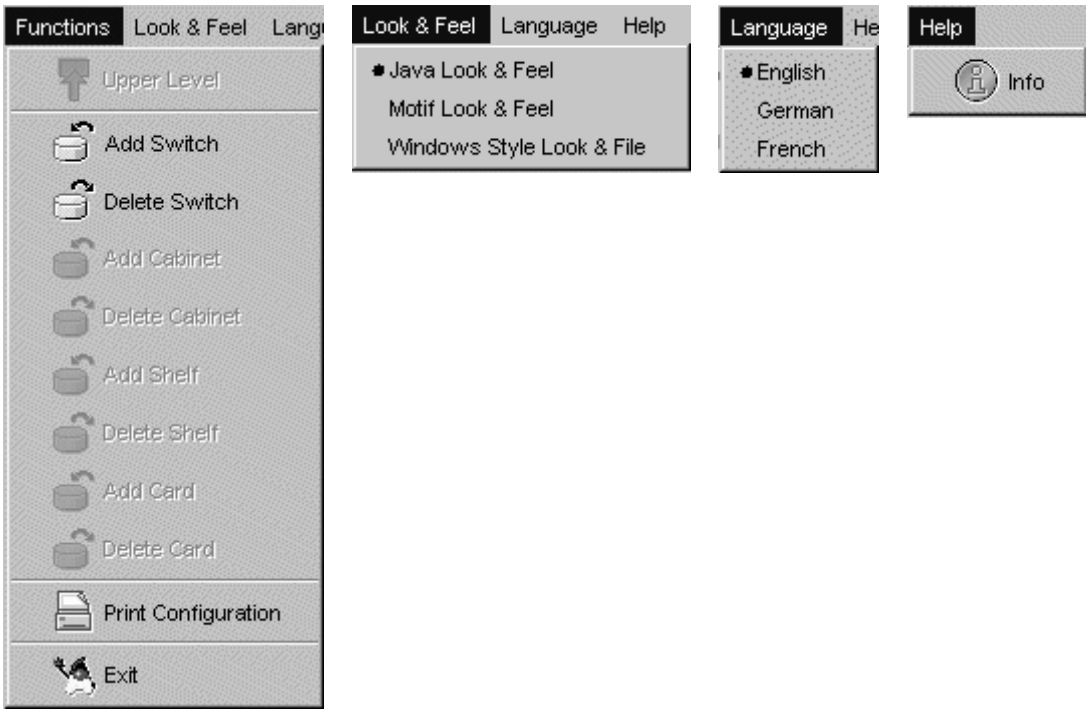


Abbildung 33: Screenshot der MenuBar auf Switch-Ebene

Abbildung 34: Screenshot der ToolBar auf Switch-Ebene

Lediglich die Möglichkeit der Aktivierung der einzelnen Schaltelemente ändert sich mit Wechsel in andere Ebenen. Diese werden entsprechend grau schattiert bzw. farbig dargestellt.

Dies ist notwendig, um auf anderen Ebenen das Hinzufügen falscher Facilities auszuschliessen. Z.B. darf es nicht möglich sein, auf Cabinet-Ebene einen Switch hinzuzufügen. Die Panels hingegen stellen dynamische Bereiche der Benutzeroberfläche dar und sind nicht dauerhaft sichtbar. Je nach Ebene wechseln diese in einen sichtbaren bzw. unsichtbaren Zustand.

Auf Ebene der Switches involviert das Panel *AvailableSwitches* (3) die Sub-Panels *PanelSwitchImage* (4)*, PanelSwitchList* (5) und *PanelSwitchDetails* (6). Diese wiederum präsentieren die Ebene in Form einer Grafik, die verfügbaren Switches innerhalb einer Liste sowie die Attribute des jeweiligen selektierten Switches.

Zu den Attributen eines Switches gehören:

- Name
- Owner
- Booking Prime
- Project
- Other Projects
- Purchasing Year

Die nachstehende Grafik veranschaulicht den Aufbau näher:

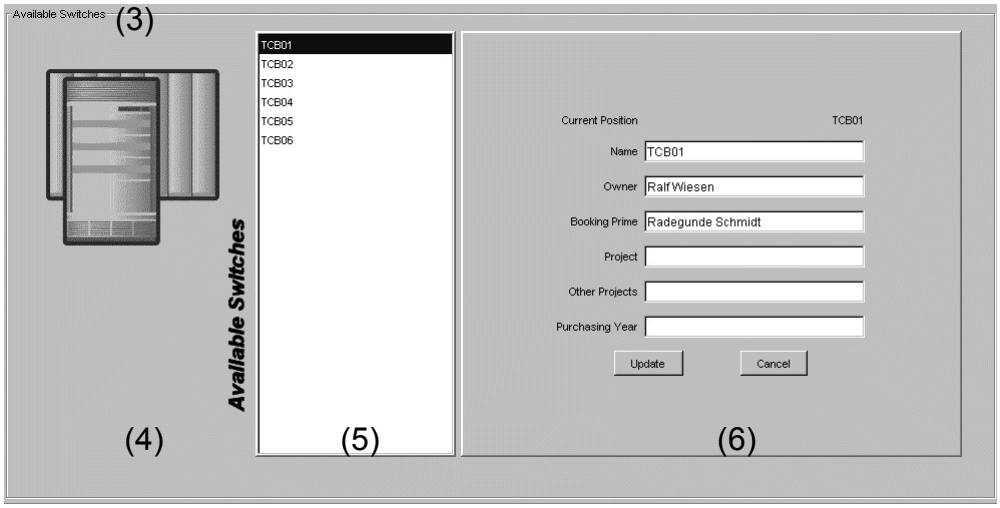


Abbildung 35: Screenshot des Panels AvailableSwitches

Das Hinzufügen von Elementen zur Liste bzw. die Entfernung dieser aus der Liste erfolgt über die in der *ToolBar* und *MenuBar* bereitgestellten Funktionen *Add Switch* bzw. *Delete Switch*. Wird die Funktion *Add Switch* aufgerufen, so öffnet sich ein Dialogfeld und der Benutzer wird gebeten, dem einzufügenden Switch einen Namen zuzuteilen.

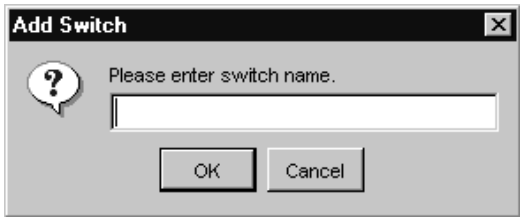


Abbildung 36: Screenshot des Dialogfeldes AddSwitch

Erst mit Betätigung der *OK*-Taste wird der Switch in die Liste und in die Datenbank aufgenommen..
Wird ein Listenelement selektiert und die Funktion *Delete Switch* aufgerufen, so erfolgt eine Sicherheitsabfrage in Form eines weiteren Dialogfeldes, um ein versehentliches Löschen eines Switches zu verhindern.

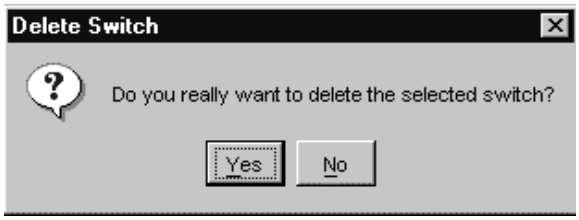


Abbildung 37: Screenshot des Dialogfelds DeleteSwitch

Auch hier erfolgt erst mit Betätigung der *OK*-Taste die gewünschte Operation in der Liste und Transaktion in der Datenbank.
Zu beachten ist, dass mit der Entfernung eines Switches ebenfalls alle hiervon abhängigen Elemente auf tieferen Ebenen entfernt werden. Wird beispielsweise ein Switch der Liste bzw. Datenbank entnommen, so werden ebenfalls alle diesem Switch zugeordneten Cabinets, Shelves und Cards gelöscht und aus der Datenbank entfernt. Um Modifikationen bezüglich Attributwerten eines Switches vornehmen zu können, ist es erforderlich, die Attributwerte über Textfelder modifizierbar zu machen. Werden Änderungen an den Attributwerten vorgenommen, so liegen diese zuerst nur in lokaler Form auf dem Client vor. Erst mit Auslösen des *Update*-Buttons unterhalb der Textfelder wird eine Transaktion ausgelöst und die modifizierten Daten in die Datenbank übernommen. Wird versehentlich eine Änderung vorgenommen, die Transaktion jedoch noch nicht ausgelöst, so besteht die Möglichkeit, über den *Cancel*-Button diese rückgängig zu machen.

Wird ein Switch in der Liste selektiert, so werden alle zu diesem gehörigen Attributwerte aus der Datenbank ausgelesen und in den im Panel *PanelSwitchDetails* vorgesehenen Textfeldern angezeigt. Erfolgt mit der linken Maustaste ein Doppel-Click auf einen Switch in der Liste, so wird in die nächst tiefere Ebene gewechselt. Bei diesem Vorgang wird das Panel *AvailableSwitches* mit seinen involvierten Panels in den unsichtbaren Zustand und ein neues Panel mit dem Namen *AvailableCabinets* in den sichtbaren Zustand gesetzt. Der Benutzer befindet sich nun auf Cabinet-Ebene und erhält alle dem Switch zugehörigen Cabinets mit ihren Attributen dargestellt.

Zu den Attributen eines Cabinets zählen:

- Name
- Number
- Sharing Switch
- Place

Erfolgt auf Switch-Ebene mit der linken Maustaste ein Doppel-Click auf ein Listenelement, ohne dass ein Cabinet existiert, das diesem Switch zugeordnet ist, so wird der Benutzer über ein Dialogfeld aufgefordert, ein neues Cabinet in der Datenbank zu erzeugen und diesem mittels dem in dem Dialogfeld vorgesehenen Textfeld einen Namen zuzuteilen.

Abbildung 38: Screenshot des Dialogfelds AddFirstCabinet

Anschliessend wird auf die Cabinet-Ebene gewechselt.

Die Applikation auf Cabinet-Ebene hat folgendes Erscheinungsbild:

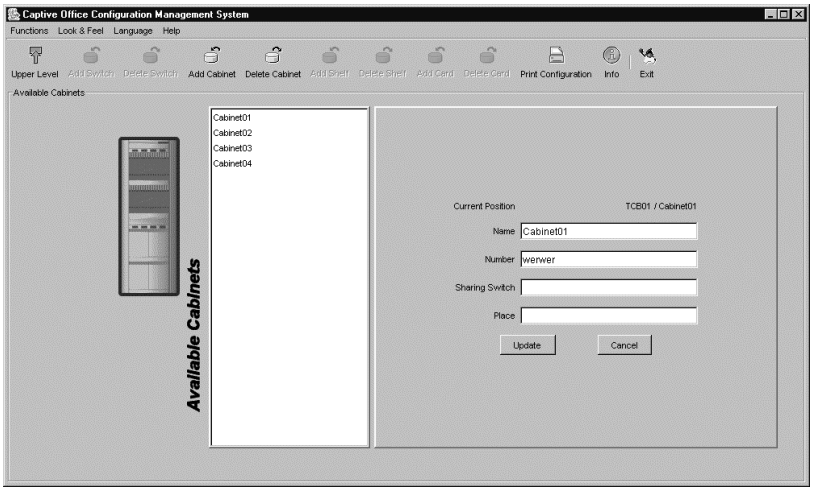


Abbildung 39: Screenshot der Cabinet-Ebene

Diese Ebene unterscheidet sich nicht wesentlich in ihrer Funktionalität von der Switch-Ebene. Lediglich die *MenuBar* und *ToolBar* ändern sich in der Aktivierung ihrer Schaltflächen. In diesen werden nun die Funktionen der Hinzufügung und Entfernung von Cabinets aktiviert und die der von Switches deaktiviert. Des Weiteren wird die Funktion des Wechseln in die nächst höhere Ebene (Switch-Ebene) zur Verfügung gestellt und somit eine Rücknavigation ermöglicht.
Die weiteren Ebenen der Verwaltung von Konfigurationsdaten für Shelves und Cards haben ähnliche Funktionalitäten wie eben genannte, lediglich differieren deren Attribute.

Zu den Attributen eines Shelfs zählen:

- Name
- Number

Zu den Attributen einer Card zählen:

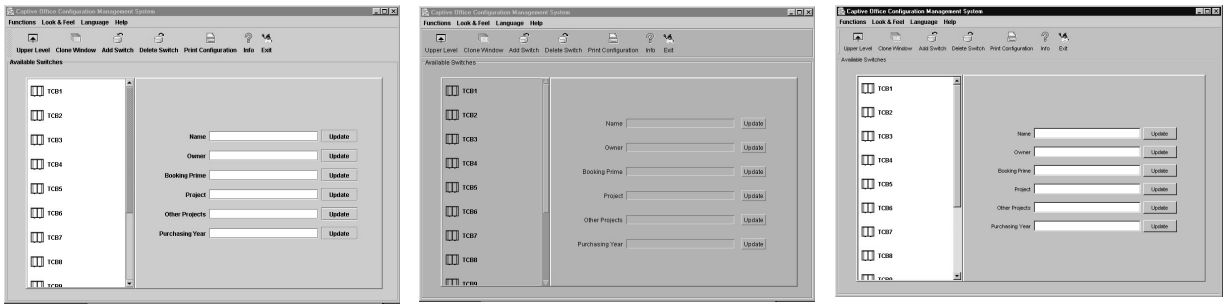
- PEC-Code
- MD-Date
- Replacement
- Hardware-Release
- Liability
- Connection
- Spare-Number
- Software-Release
- Frontpanel / Backpanel

Um die Möglichkeit zu schaffen, die Konfigurationsdaten einer Facility zu drucken, wird hierfür eine Druck-Funktion bereitgestellt, die sowohl über die *MenuBar* als auch über die *ToolBar* aktiviert werden kann. Diese druckt die jeweils selektierte Facility mit ihren zugehörigen Attributen.

Des Weiteren ist es möglich über die Menüfunktion *Language* in der *MenuBar* die Applikation bezüglich ihrer Sprache anzupassen. Man spricht hier von der *Internationalization*, die es ebenfalls erlaubt, während der Laufzeit der Anwendung die Sprache zu variieren.

Zudem ist auch eine Variation in Bezug auf das äussere Erscheinungsbild der Applikation möglich. Hierbei handelt es sich um das bereits angesprochene *Pluggable Look & Feel*, das es möglich macht, die Anwendung in unterschiedlichem Gewand erscheinen zu lassen. Zur Auswahl existieren neben dem *Windows*- noch das *Motif*- und *Metal-Look & Feel*. Mit dem Start der Anwendung wird an erster Stelle überprüft, welche Art von Plattform vorliegt und das *Look & Feel* der Anwendung entsprechend adaptiert. Hinzu kommt, dass das Umschalten des *Look & Feels* ebenfalls während der Laufzeit der Applikation möglich ist, ohne die Applikation schliessen zu müssen.

Die folgende Abbildung zeigt den Prototyp der entwickelten Anwendung in unterschiedlichen *Look & Feels*:



Java Look & Feel *Motif Look & Feel* *Windows Look & Feel*

Abbildung 40: Unterschiedliche Look & Feels einer Applikation

4.3 Modellierung der Datenbank

Mit dem Starten der Applikation auf dem Client wird eine Verbindung zur Datenbank hergestellt, die während der gesamten Laufzeit der Anwendung bestehen bleibt. Bei der verwendeten Datenbank handelt es sich um *PostgreSQL*, welche eine leistungsfähige, relationale Datenbank für die Abfragesprache *SQL* darstellt, inklusive Quellcode frei erhältlich ist und ohne Lizensierung auch im kommerziellen Umfeld benutzt werden darf. Diese befindet sich auf einem zentralen Webserver und wird so konfiguriert, dass sie von jedem Client mit der IP-Adresse 131.147.xxx.xxx zugänglich ist. Dies erfolgt mittels dem folgenden Eintrag in der Konfigurationsdatei "pg_hba.conf" der Datenbank:

host	*comansys*	*131.147.0.0*	*255.255.0.0*	*trust*
(Typ)	(Datenbankname)	(IP-Adresse)	(Subnetzmaske)	(Auth.Typ)

Anschliessend ist es erforderlich, mit dem Kommando
postmaster -i -D /var/lib/pqsql/data den Datenbank-Prozess neu zu starten. Der Parameter *-i* sorgt dafür, dass TCP/IP-Verbindungen (Transmission Control Protocol / Internet Protocol) zum Datenbankserver zugelassen werden. Der Parameter *-D* spezifiziert den Pfad zur Datenbank.

Bezüglich der Modellierung der Datenbank wird das Design an die vorliegende hierarchische Anordnung der Hardware angelehnt und somit vier Relationen erstellt, in welchen die Daten für die jeweiligen Facilities abgelegt werden. Diese erhalten der Einfachheit halber dieselben Namen wie die zu verwaltenden Facilities und beinhalten die bereits auf den vorigen Seiten erwähnten Attribute. Bei dieser Art der Modellierung liegt bereits eine Normalisierung vor, da jegliche Redundanz von Daten mit der Referenzierung der Relationen beseitigt wird. Keine der Relationen weist in irgendeinem Bereich identische Werte auf.

Die folgenden Abbildungen veranschaulichen den Aufbau der vier Relationen:

Field	Type	Length	Not Null	Default
pk_switch	int4	4	Yes	nextval('"switch_pk_switch_seq"'::text)
switchname	varchar	100	No	
switchowner	varchar	100	No	
switchbookingprime	varchar	100	No	
switchproject	varchar	100	No	
switchotherprojects	varchar	100	No	
switchpurchasingyear	varchar	100	No	

Abbildung 41: Datenmodellierung der Switches

Field	Type	Length	Not Null	Default
pk_cabinet	int4	4	Yes	nextval('"cabinet_pk_cabinet_seq"'::text)
cabinetname	varchar	100	No	
cabinetnumber	varchar	100	No	
cabinetsharingswitch	varchar	100	No	
cabinetplace	varchar	100	No	
cabinet_fkswitch	int4	4	Yes	

Abbildung 42: Datenmodellierung der Cabinets

Field	Type	Length	Not Null	Default
pk_shelf	int4	4	Yes	nextval('"shelf_pk_shelf_seq"'::text)
shelfname	varchar	100	No	
shelfnumber	varchar	100	No	
shelf_fkcabinet	int4	4	Yes	

Abbildung 43: Datenmodellierung der Shelves

Field	Type	Length	Not Null	Default
pk_card	int4	4	Yes	nextval('"card_pk_card_seq"'::text)
cardpeccode	varchar	100	No	
cardmddate	varchar	100	No	
cardreplacement	varchar	100	No	
cardhardwarerelease	varchar	100	No	
cardliability	varchar	100	No	
cardconnection	varchar	100	No	
cardsparenumber	varchar	100	No	
cardsoftwarerelease	varchar	100	No	
cardpanelside	varchar	100	No	
card_fkshelf	int4	4	Yes	

Abbildung 44: Datenmodellierung der Cards

Allen vier Relationen lässt sich entnehmen, dass mit Ausnahme des Primärschlüssels die verwendeten Attribute alle der gleichen Domäne entstammen, vom Typ *varchar* sind und eine maximale Länge von 100 Zeichen besitzen. Des Weiteren lässt sich erkennen, dass jeder der vier Relationen ein zusätzliches Attribut vom Typ *int4* mit dem Namen *pk_relationsname* hinzugefügt wird. Dieses entspricht einem *Integer*-Wert mit einer Länge von 4 Byte und sorgt als Primärschlüssel der Relation für die eindeutige Identifizierung der Tupel. Zudem beinhalten die Relationen *Cabinet, Shelf* und *Card* ein weiteres Attribut am Ende der Relation, welches als Fremdschlüssel und somit der Referenzierung auf andere Relationen dient. Dieses ist vom Typ *int4* und beinhaltet Werte des Primärschlüssels der referenzierten Relation. Bei dieser Art der Referenzierung handelt es sich um eine "1:n"-Verknüpfung, da beispielsweise genau ein Cabinet mehrere Shelves beinhalten kann.

Zu beachten ist, dass die Felder des Primär- und Fremdschlüssels niemals Nullwerte annehmen dürfen, da sonst keine eindeutige Identifizierung bzw. Referenzierung gewährleistet werden kann.

Die folgende Abbildung zeigt beispielhaft die verwendeten Relationen und deren Referenzierung:

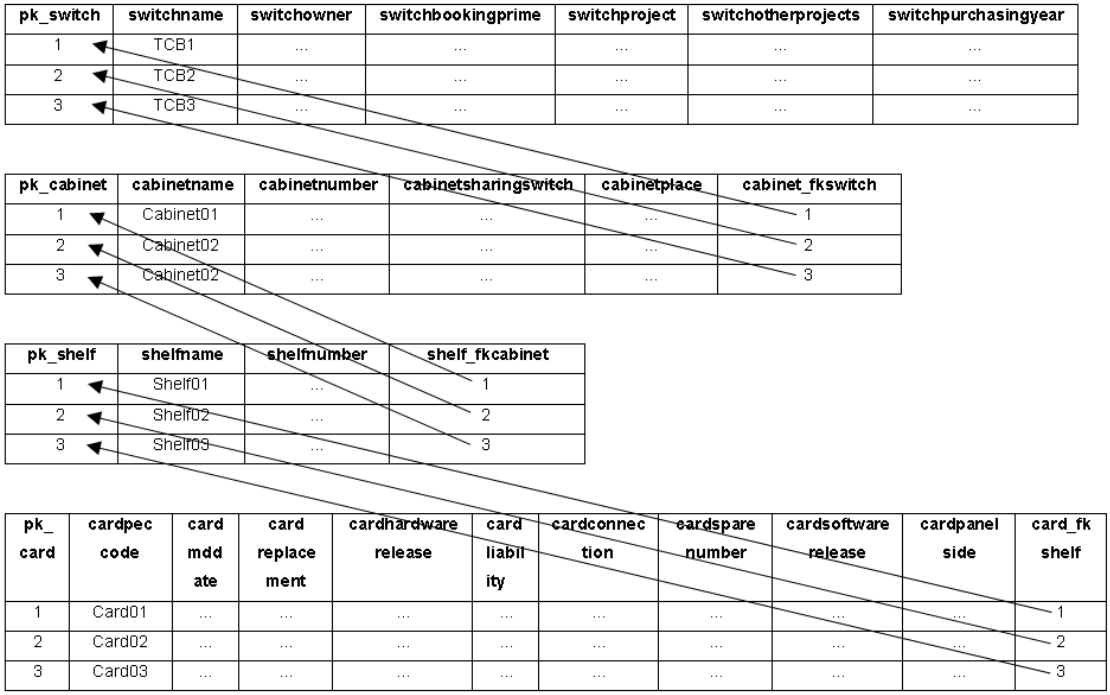


Abbildung 45: Relationen und deren Referenzen

Um Inkonsistenzen beim Zugriff auf die Datenbank ausschliessen zu können, ist es wichtig, die Wertzuweisung des Primärschlüssels auf der Server-Seite vom Datenbankmanagementsystem selbst durchführen zu lassen. Dies erfolgt über eine von der Datenbank bereitgestellte Funktion *nextval()*, die innerhalb der Relation jedem neu hinzugefügten Element als Primärschlüssel einen *Integer*-Wert mit einer maximalen Länge von 4 Byte zuweist. Hierbei wird mit dem Wert 1 begonnen und bei jedem Einfügevorgang dieser Wert um 1 inkrementiert. Somit wird eine eindeutige Identifizierung der Tupel innerhalb der Relation gewährleistet.
Des Weiteren ist zu berücksichtigen, dass mit dem Entfernen eines Listenelements alle hiervon abhängigen Elemente auf tieferen Ebenen entfernt werden müssen. Wird beispielsweise ein Cabinet der Liste entnommen, so müssen ebenfalls alle diesem Cabinet zugeordneten Shelves und Cards gelöscht werden. Dies lässt sich in *SQL* mithilfe des Constraints *ON DELETE CASCADE* realisieren, welches beim Entfernen eines Datums alle referenzierten Daten ebenfalls beseitigt. Das Constraint muss bereits bei der Erstellung und Definition einer Relation berücksichtigt werden.
Nachstehende Zeilen zeigen veranschaulichen dies:

```
CREATE TABLE "cabinet" (
"pk_cabinet" serial NOT NULL,
"cabinetname" varchar (100) NULL,
"cabinetnumber" varchar (100) NULL,
"cabinetsharingswitch" varchar (100) NULL,
"cabinetplace" varchar (100) NULL,
"cabinet_fkswitch" varchar (100) REFERENCES "switch"("pk_switch") ON DELETE
CASCADE, PRIMARY KEY (pk_cabinet) );
```

5 Implementierung

In diesem Kapitel wird die Implementierungsphase für das Captive Office Configuration Management System beschrieben und die in der Analysephase definierten Anforderungen in ihrer Umsetzung näher erläutert.
Hierbei wird anhand konkreter Anwendungsfälle auf die Realisierung genau eingegangen und die verwendeten Klassen mit ihren Attributen und Funktionen vorgestellt. Mittels Interaktionsdiagrammen und Quelltextauszügen wird erläutert, wie die Anwendungsfälle in der Programmiersprache *JAVA* realisiert werden.

5.1 Realisierung der Anwendungsfälle

5.1.1 Starten der Anwendung

Der Start der Anwendung erfolgt über die Klasse *Main*, welche die Hauptklasse der Anwendung darstellt und für die Herstellung einer Verbindung zum Datenbankmanagementsystem sowie der Initialisierung von Sprache und *Look & Feel* zuständig ist.
Der Verbindungsaufbau zur Datenbank erfolgt an erster Stelle über das Laden eines *JDBC*-Treibers vom Typ 4 (Native Protocol "Pure JAVA" Driver) mit dem Befehl

Class.forName("org.postgresql.Driver").

Anschliessend wird mittels der Methode

DriverManager.getConnection("jdbc:postgresql://131.147.37.81/" + database,
username, password)

eine Verbindung zur Datenbank hergestellt.
Ist die Verbindung hergestellt, so wird ein neues Objekt vom Typ *Locale* erzeugt und mithilfe von diesem und der Methode

ResourceBundle.getBundle("MessagesBundle", currentLocale)

ein neues *Bundle* für die initialen Spracheinstellungen geladen.

Um die initialen Einstellungen abzuschliessen, ist die Festlegung des der Plattform angelehnten *Look & Feels* notwendig. Dies erfolgt innerhalb eines *try/catch*-Blocks, wie er im Folgenden dargestellt wird:

```
try {
    UIManager.setLookAndFeel( UIManager.getSystemLookAndFeelClassName() );
    SwingUtilities.updateComponentTreeUI( frame );
} catch ( Exception e ) {}
```

Anschliessend wird durch Erzeugung eines Objektes vom Typ *Gui* die Benutzeroberfläche mit ihrer *MenuBar*, *ToolBar* sowie ihren Panels erzeugt.
Bezüglich der Panels ist zu nennen, dass lediglich das Panel *PanelSwitch* hierbei dem Inhaltsfenster über die Methode

```
contentPane.add( panelSwitch, BorderLayout.CENTER );
```

hinzugefügt wird. Alle anderen Panels existieren zwar, befinden sich jedoch in einer unsichtbaren Form im Hintergrund. Wurden mittels der Methode *createGUI()* alle Benutzeroberflächen-Elemente kreiert, so müssen diese nun mit *ActionListenern* versehen werden, damit auf Ereignisse mit der Maus entsprechend reagiert werden kann. Dies erfolgt in der Methode *addActionListener()*, die den Schaltflächen der Benutzeroberfläche die jeweiligen *ActionListener* zuweist und in Abhängigkeit vom jeweiligen Ereignis die zugeordnete Funktion aufruft. Der nachstehende Code zeigt beispielhaft die Zuordnung eines *ActionListeners* zur Schaltfläche *Add Switch*, die es ermöglicht, eine Facility vom Typ Switch der Datenbank hinzuzufügen:

```
buttonToolBarAddSwitch.addActionListener( new ActionListener() {
        public void actionPerformed( ActionEvent theEvent )
        {
                addSwitchClick();
        }
});
```

5.1.2 Hinzufügen einer Facility vom Typ Switch

Wird ein Ereignis durch einen einfachen Click mit der linken Maustaste auf die Schaltfläche *Add Switch* in der *MenuBar* oder *ToolBar* ausgelöst, so sorgt der diesen Schaltflächen zugeordnete *ActionListener* für den Aufruf der Methode *actionPerformed()*, welche die Methode *addSwitchClick()* aktiviert. Innerhalb dieser Methode wird ein Dialogfenster generiert und der Benutzer über ein *TextField* zur Eingabe eines repräsentativen Namens für den hinzuzufügenden Switch gebeten. Nachstehender Code veranschaulicht die Erstellung eines solchen Dialogfensters:

```
JOptionPane.showInputDialog( this, main.getLabelText( "addSwitchDialog" ),
                                   main.getLabelText( "addSwitch" ) ,
                                   JOptionPane.QUESTION_MESSAGE );
```

Mit Bestätigung des eingegebenen Namens über den *OK*-Button wird der String, der den Namen des eingegebenen Switches beinhaltet, der Funktion *addElement()* der Klasse *ResultSetListModel* übergeben.
Die Methode *addElement()* erzeugt an erster Stelle ein Objekt vom Typ *Statement,* welchem später das *SQL-Statement* in Form einer Zeichenkette zugewiesen wird. Anschliessend wird überprüft, um welche Art von Facility es sich handelt, da das Hinzufügen eines Switches im Gegensatz zu einem Cabinet, einem Shelf oder einer Card nicht fremdschlüsselbehaftet ist und sich somit das *SQL-Statement* in seinem Aufbau unterscheidet. Handelt es sich um eine Facility vom Typ Switch, so wird das *SQL-Statement* folgendermassen generiert:

```
String sqlText = "INSERT INTO " + tableName + " ( switchname )" + "VALUES
                                                ( '" + elementName.toString() + "' )";
```

Anschliessend wird das dem String *sqlText* zugewiesene *Statement* über die Methode

```
sql.executeUpdate( sqlText );
```

ausgeführt.

Das Objekt wird nun der Datenbank hinzugefügt. Der Ergebnisdatensatz auf dem Client entspricht jedoch nicht mehr dem der Datenbank. Aufgrund dieser Tatsache ist es erforderlich, nach diesem Vorgang den Ergebnisdatensatz der Anwendung mit dem der Datenbank zu synchronisieren.
Hierfür muss aufgrund des fehlenden Fremdschlüssels bei Facilities vom Typ Switch wiederum eine Fallunterscheidung gemacht werden und das *ResultSet* schliesslich mittels dem Befehl

```
rs = stmt.executeQuery( "SELECT * FROM " + tableName + " ORDER BY
                                                    switchname" );
```

aktualisiert werden. Mittels der *SQL*-Anweisung *ORDER BY SWITCHNAME* ist es möglich, Ergebnisse einer Anfrage alphabetisch sortieren zu lassen.
Auch für diese Anfrage an die Datenbank muss explizit ein Objekt vom Typ *Statement* erzeugt werden, welches in diesem Fall den Namen *stmt* trägt.
Da sich an dieser Stelle Änderungen bezüglich dem zu überwachenden *Subject* ergeben, ist es nun erforderlich, die Grösse des Ergebnisdatensatzes neu zu bestimmen sowie ein Ereignis zur Benachrichtigung der registrierten *Observer* zu senden. Dies erfolgt anhand der folgenden Zeilen im Code:

```
rs.last();                  // Sprung an das letzte Element im ResultSet
setSize( rs.getRow() );     // Ermittlung des Index des letzten Elements im ResultSet
                               und neue Bestimmung der Grösse

fireIntervalAdded( this, index, index+1 );        // Ereignisauslösung
                                        (Name der Property, alter Wert, neuer Wert)
```

Alle genannten Operationen müssen in einem *try/catch*-Block ausgeführt werden; im Falle einer Datenbankzugriffsverletzung wird eine *SQL-Exception* geworfen.
Die Ereignisauslösung über die Methode *fireIntervalAdded()* bewirkt, dass in den angemeldeten *Listenern,* hier *PanelSwitchDetails,* die Methode *propertyChange()* aufgerufen wird, welche eine Aktualisierung der Benutzeroberfläche veranlässt.

Die Aktualisierung der Benutzeroberfläche in der Methode *propertyChange()* erfolgt mittels folgender Befehle:

```
text = listModel.getString("switchname");
if ( text != null )
        textFieldName.setText( text );
else
        textFieldName.setText( "" );
.....
.....
```

Hierbei werden alle *TextFields*, die die Attributwerte der Facility repräsentieren, neu gesetzt.

Das nachstehende Sequenzdiagramm veranschaulicht dies näher:

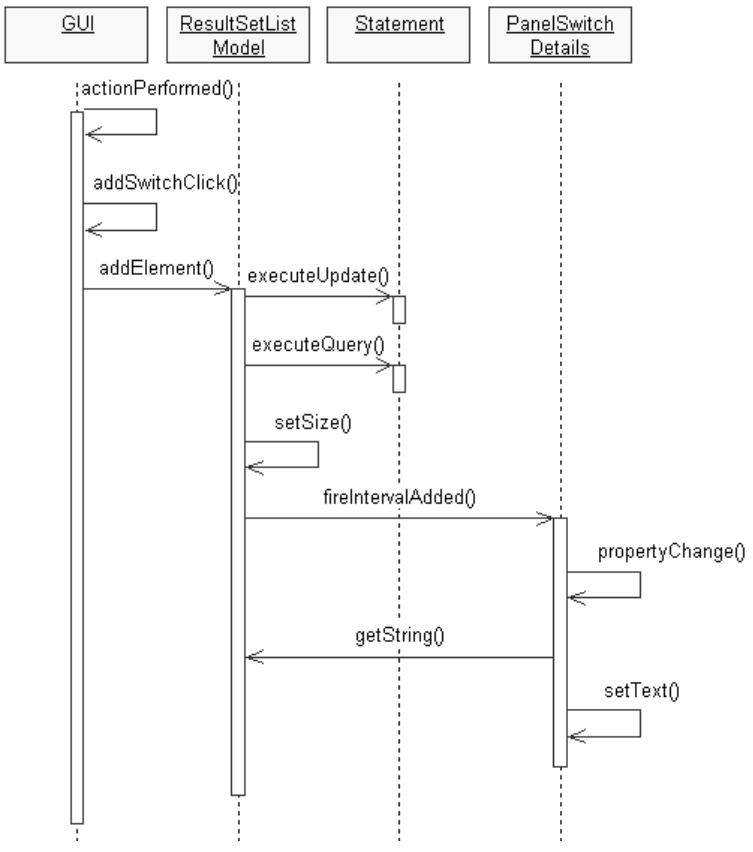


Abbildung 46: Sequenzdiagramm der Hinzufügung eines Switches

5.1.3 Entfernen einer Facility vom Typ Switch

Das Ereignis der Entfernung eines Switches wird durch Selektion des jeweiligen Listenelements und einem einfachen Click mit der linken Maustaste auf die Schaltfläche *Delete Switch* in der *MenuBar* oder *ToolBar* ausgelöst.
Der diesen Schaltflächen zugeordnete *ActionListener* sorgt anschliessend für den Aufruf der Methode *actionPerformed()*, welche wiederum die Methode *delSwitchClick()* aktiviert. Diese Methode generiert ebenfalls ein Dialogfenster, in welchem der Benutzer aufgefordert wird, seine gewünschte Aktion zu bestätigen.
Der nachstehende Code repräsentiert die Erstellung eines solchen Dialogfensters:

```
JOptionPane. showConfirmDialog( this, main.getLabelText( "deleteSwitchDialog" ),
                                      main.getLabelText( "deleteSwitch" ) ,
                                      JOptionPane.YES_NO_OPTION,
                                      JOptionPane.QUESTION_MESSAGE );
```

Mit Bestätigung durch den *YES*-Button wird über die von der Klasse *ResultSetListModel* bereitgestellte Funktion *getIndex()* der Index des selektierten Listenelements ermittelt und anschliessend die Methode *remove()* mit dem ermittelten Index als Übergabeparameter aufgerufen. Die Methode *remove()* erzeugt wie auch die Methode *addElement()* an erster Stelle ein Objekt vom Typ *Statement,* welchem später das *SQL-Statement* in Form einer Zeichenkette zugewiesen wird. Anschliessend wird überprüft, um welche Art von Facility es sich handelt, da das Entfernen eines Switches im Gegensatz zu einem Cabinet, einem Shelf oder einer Card nicht fremdschlüsselbehaftet ist und sich somit das *SQL-Statement* in seinem Aufbau unterscheidet. Handelt es sich um eine Facility vom Typ Switch, so wird das *SQL-Statement* folgendermassen generiert:

```
String sqlText = "DELETE FROM " + tableName + " WHERE " + pkFieldName + " = '"
                                    + ( String )rs.getString( "pk_switch" ) + "'";
```

Das obenstehende *SQL-Statement* zeigt, dass bei diesem Vorgang der Primärschlüssel für die eindeutige Identifizierung benötigt wird.

Dieser kann über die Funktion

rs.getString("pk_switch")

aus dem Ergebnisdatensatz ermittelt werden.
Anschliessend wird das dem String *sqlText* zugewiesene *Statement* über die Methode

sql.executeUpdate(sqlText);

ausgeführt. Das Objekt wird nun aus der Datenbank entfernt, der Ergebnisdatensatz auf dem Client entspricht jedoch nicht mehr dem der Datenbank. Aus diesem Grund ist es auch hier erforderlich, nach dem Entfernen eines Elements den Ergebnisdatensatz der Anwendung mit dem der Datenbank zu synchronisieren.
Aufgrund des fehlenden Fremdschlüssels bei Facilities vom Typ Switch muss wiederum eine Fallunterscheidung gemacht werden und das *ResultSet* schliesslich mittels dem Befehl

*rs = stmt.executeQuery("SELECT * FROM " + tableName + " ORDER BY switchname");*

aktualisiert werden. Für diese Anfrage an die Datenbank muss explizit ein Objekt vom Typ *Statement* erzeugt werden, welches in diesem Fall den Namen *stmt* trägt.
Auch an dieser Stelle werden mittels der *SQL*-Anweisung *ORDER BY switchname* die Ergebnisse der Anfrage alphabetisch sortiert. Durch die Entfernung einer Facility ergeben sich Änderungen bezüglich dem zu überwachenden *Subject*. Darum ist es erforderlich, im Anschluss daran die Grösse des Ergebnisdatensatzes neu zu bestimmen sowie ein Ereignis zur Benachrichtigung der registrierten *Observer* zu senden.

Dies erfolgt anhand der folgenden Zeilen im Code:

```
int i = getSize();                              // Ermittlung der bisherigen Grösse
setSize( --i );                                 // Dekrementierung der Grösse um 1
fireIntervalRemoved( this, index, index-1);     // Ereignisauslösung
                                                (Name der Property, alter Wert, neuer Wert)
```

Alle genannten Operationen müssen in einem *try/catch*-Block ausgeführt werden. Im Falle einer Datenbankzugriffsverletzung wird eine *SQL-Exception* geworfen.
Die Ereignisauslösung über die Methode *fireIntervalRemoved()* bewirkt, dass in den angemeldeten *Listenern,* hier *PanelSwitchDetails,* die Methode *propertyChange()* aufgerufen wird, welche eine Aktualisierung der Benutzeroberfläche veranlässt.

Die Aktualisierung der Benutzeroberfläche in der Methode *propertyChange()* erfolgt mittels folgender Befehle:

```
text = listModel.getString("switchname");
if ( text != null )
        textFieldName.setText( text );
else
        textFieldName.setText( "" );
.....
.....
```

Hierbei werden alle *TextFields*, die die Attributwerte der Facility repräsentieren, neu gesetzt.

Das nachstehende Sequenzdiagramm veranschaulicht den beschriebenen Vorgang:

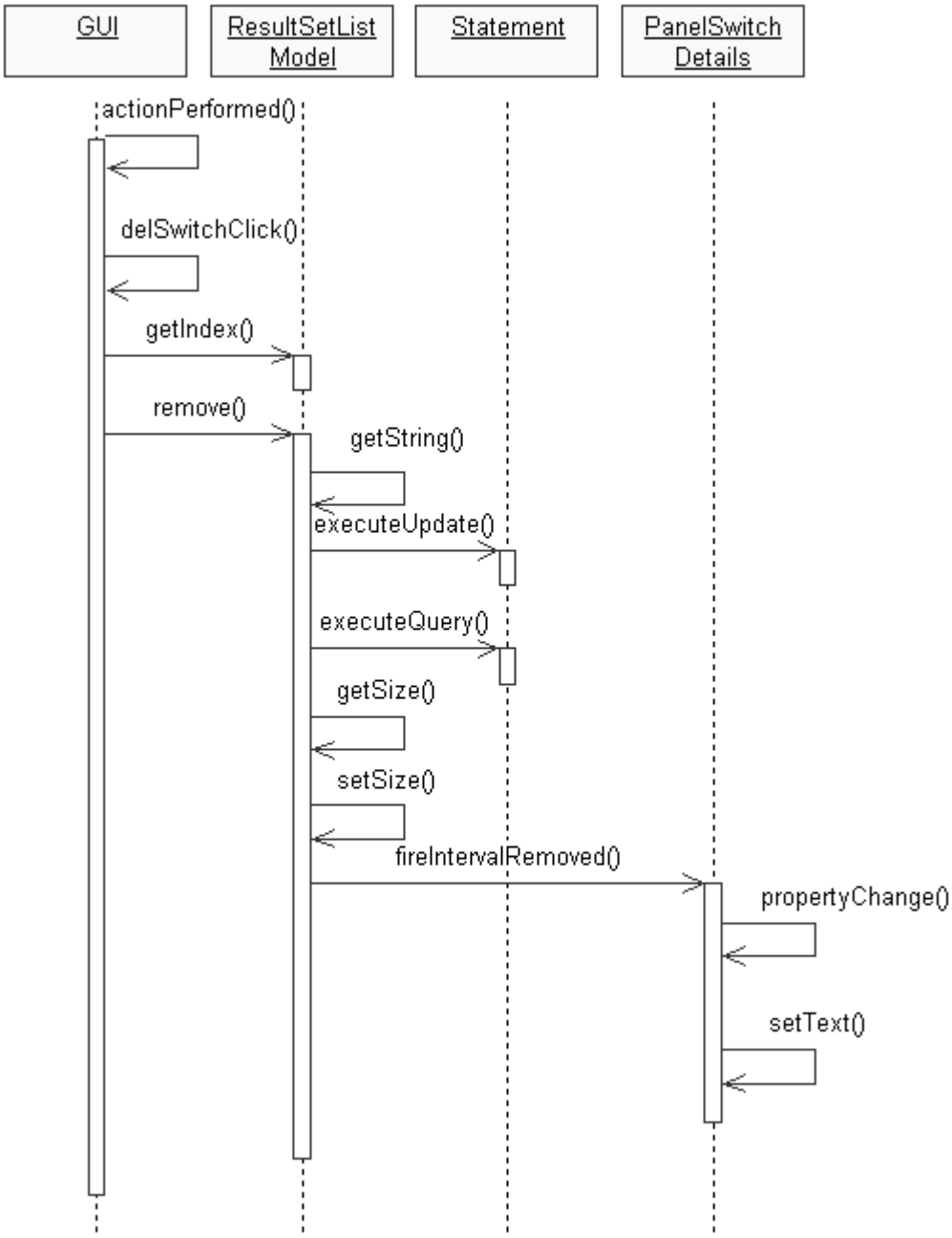


Abbildung 47: Sequenzdiagramm der Entfernung eines Switches

5.1.4 Hinzufügen einer Facility vom Typ Cabinet/Shelf/Card

Das Hinzufügen einer Facility vom Typ Cabinet, Shelf oder Card unterscheidet sich nicht wesentlich von dem eines Switches. Lediglich die *SQL-Statements* beim Einpflegen in die Datenbank bzw. Auslesen aus dieser unterscheiden sich in ihrem Aufbau. Für das Hinzufügen eines Cabinets hat das *SQL-Statement* beispielsweise folgenden Aufbau:

```
String sqlText = "INSERT INTO " + tableName + " ( " + fkFieldName + ", cabinetname )" + "VALUES ('" + fkFieldValue + "', '" + elementName.toString() + "')";
```

Dem obigen *SQL-Statement* ist zu entnehmen, dass für diesen Vorgang ein Fremdschlüssel benötigt wird, der die Facility in Abhängigkeit von ihrer Referenzierung in die Datenbank einpflegt.

Anschliessend wird das dem String *sqlText* zugewiesene *Statement* über die Methode

```
sql.executeUpdate( sqlText );
```

ausgeführt. Das Objekt wird nun der Datenbank hinzugefügt, der Ergebnisdatensatz auf dem Client muss jedoch auch an dieser Stelle mit dem der Datenbank synchronisiert werden. Hierzu dient das folgende *SQL-Statement*:

```
rs = stmt.executeQuery( "SELECT * FROM " + tableName + " WHERE " + fkFieldName + " = '" + fkFieldValue + "' ORDER BY cabinetname" );
```

All übrigen Anweisungen gleichen denen der Hinzufügung eines Switches. Lediglich die Behaftung der Daten mit einem Fremdschlüssel muss berücksichtigt werden.

5.1.5 Entfernen einer Facility vom Typ Cabinet/Shelf/Card

Die Entfernung einer Facility vom Typ Cabinet, Shelf oder Card lehnt sich der eines Switches an und unterscheidet sich nahezu nicht in ihrem Ablauf, da innerhalb der Relation eine eindeutige Vergabe des Primärschlüssels stattfindet und der Vorgang der Entfernung allein mittels Indentifizierung durch diesen vonstatten gehen kann.
Hierbei muss der Fremdschlüssel nicht berücksichtigt werden.
Das *SQL-Statement* für die Entfernung eines Cabinets hat beispielsweise folgenden Aufbau:

String sqlText = "DELETE FROM " + tableName + " WHERE " + pkFieldName + " = '" + (String)rs.getString(pkFieldName) + "';

Anschliessend wird das dem String *sqlText* zugewiesene *Statement* über die Methode

sql.executeUpdate(sqlText);

ausgeführt.

Das Objekt wird nun aus der Datenbank entfernt, der Ergebnisdatensatz auf dem Client muss jedoch auch an dieser Stelle mit dem der Datenbank synchronisiert werden. Hierzu dient das folgende *SQL-Statement*:

*rs = stmt.executeQuery("SELECT * FROM " + tableName + " WHERE " + fkFieldName + " = '" + fkFieldValue + "' ORDER BY " + pkFieldName);*

All übrigen Anweisungen gleichen denen der Entfernung eines Switches. Lediglich bei der Synchronisierung muss die Behaftung der Daten mit einem Fremdschlüssel berücksichtigt werden.

5.1.6 Einpflegen modifizierter Daten in die Datenbank

Werden Modifikationen bezüglich den Attributwerten von Facilities vorgenommen, so kann mittels der Schaltfläche *Update* eine Transaktion ausgelöst und die Daten in die Datenbank übernommen werden. Mit dem Auslösen eines Ereignisses dieser Art wird in der Klasse *PanelXXXDetails* eine *HashMap* (Tabelle) erzeugt, welche alle in den *TextFields* vorhandenen Daten einliest. Im Anschluss darauf wird die Methode *rowUpdate()* der Klasse *ResultSetListModel* aufgerufen und die *HashMap* dieser übergeben. Der folgende Code veranschaulicht diesen Vorgang näher:

```
table = new HashMap();
table.put("switchname", textFieldName.getText() );
table.put("switchowner", textFieldOwner.getText() );
table.put("switchbookingprime", textFieldBookingPrime.getText() );
table.put("switchproject", textFieldProject.getText() );
table.put("switchotherprojects", textFieldOtherProjects.getText() );
table.put("switchpurchasingyear", textFieldPurchasingYear.getText() );

((ResultSetListModel)listModel).rowUpdate(table);
```

Die Methode *rowUpdate()* realisiert den Datenbankzugriff und ermöglicht es, mittels Erzeugung eines *Iterators* die *HashMap* zu durchlaufen und innerhalb einer *while*-Schleife diese Daten in die Datenbank einzupflegen.
Nachstehende Zeilen veranschaulichen diesen Vorgang:

```
Iterator rowWalker = row.keySet().iterator();
Statement sql = main.con.createStatement();
while ( rowWalker.hasNext() ) {
        String fieldName = (String)rowWalker.next();
        String sqlText = "UPDATE " + tableName + " SET " + fieldName
                + " = '" + (String)row.get( fieldName ) + "' WHERE " +
                pkFieldName + " = '" + (String)rs.getString(pkFieldName) + "'";
        sql.executeUpdate(sqlText);
}
```

Im Anschluss ist aufgrund modifizierter Daten in der Datenbank eine Aktualisierung des clientseitigen Ergebnisdatensatzes notwendig. Dies erfolgt abhängig von der jeweiligen Facility wie bereits in Kapitel 5.1.2 beschrieben. Auch an dieser Stelle ist es wieder erforderlich, die registrierten *Listener* über die Modifikation der Daten zu benachrichtigen. Hierzu dient die Methode *fireContentsChanged()*, welche eine Zustandsänderung signalisiert und eine Aktualisierung der Benutzeroberflächen mittels der Methode *propertyChange()* veranlässt.
Das folgende Sequenzdiagramm veranschaulicht diesen Vorgang:

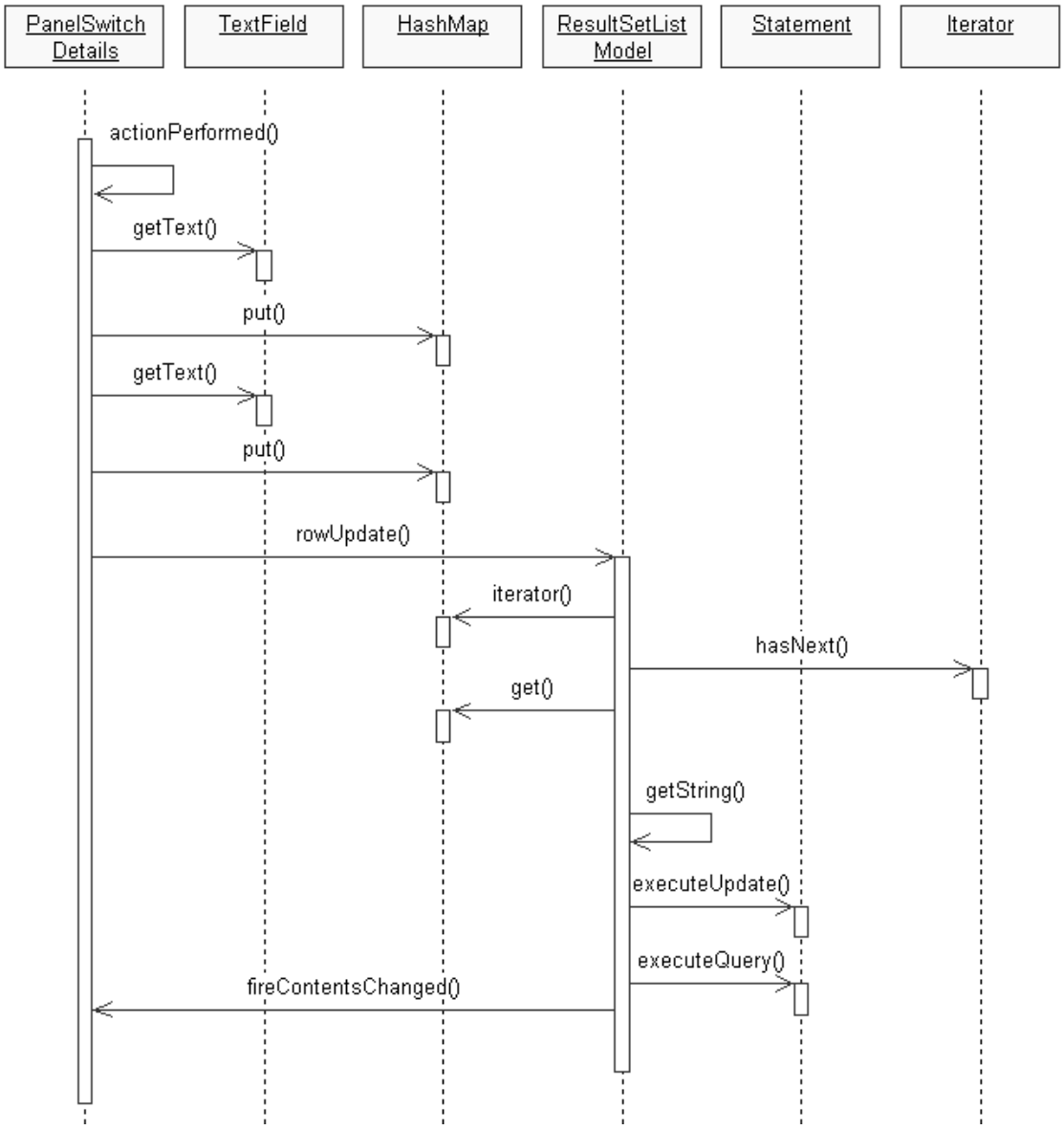


Abbildung 48: Sequenzdiagramm der Modifikation von Attributen

5.1.7 Modifikationen rückgängig machen

Werden Attributwerte versehentlich modifiziert, jedoch noch nicht mittels einer Transaktion in die Datenbank übernommen, so besteht die Möglichkeit mittels der Schaltfläche *Cancel* diese wieder rückgängig zu machen. Hierbei werden lediglich die Werte des auf der Client-Seite vorliegenden Ergebnisdatensatzes wieder zurück in die *TextFields* geschrieben, welche die Attributwerte der Facility repräsentieren.
Dies erfolgt unmittelbar in der Methode *actionPerformed()*, welche bei Ereignisauslösung durch die Schaltfläche *Cancel* aufgerufen wird.
Die nachstehenden Zeilen zeigen diesen Vorgang:

```
textFieldName.setText(listModel.getString("switchname"));
textFieldOwner.setText(listModel.getString("switchowner"));
textFieldBookingPrime.setText(listModel.getString("switchbookingprime"));
textFieldProject.setText(listModel.getString("switchproject"));
textFieldOtherProjects.setText(listModel.getString("switchotherprojects"));
textFieldPurchasingYear.setText(listModel.getString("switchpurchasingyear"));
```

5.1.8 Selektion einer Facility in der Liste

Alle verfügbaren Facilities einer Art werden auf der jeweiligen Ebene in Form einer Liste präsentiert. Um die jeweiligen Attributwerte des selektierten Listenelements im rechts davon befindlichen Panel anzeigen zu können, müssen diese in Abhängigkeit des Index der Liste aus dem Ergebnisdatensatz ausgewählt werden.
Hierfür wird der Liste mit der Methode *addListSelectionListener()* ein *Listener* hinzugefügt, der bei wechselnder Selektion von Listenelementen die Methode *valueChanged()* aufruft. In dieser Methode wird mittels dem Aufruf

```
((ResultSetListModel)listModel).setIndex(list.getSelectedIndex() );
```

der Index des Ergebnisdatensatzes entsprechend der Listenselektion neu gesetzt. Die Methode *setIndex()* der Klasse *ResultSetListModel* wiederum aktiviert die Methode *firePropertyChange()*, welche eine Zustandsänderung des Index signalisiert und die Benutzeroberflächen mittels der Methode *propertyChange()* zu einer Aktualisierung veranlässt.

5.1.9 Wechsel in eine tiefere Ebene

Um alle einer Facility zugehörigen Elemente der nächst tieferen Ebene mit ihren Attributen anzuzeigen, bedarf dies einem doppelten Click mit der linken Maustaste auf das jeweilige Listenelement. Hierfür wird der Liste neben dem *Listener* für die Selektion von Listenelementen ein weiterer *Listener* mit der Methode *addMouseListener()* hinzugefügt. Dieser registriert Clicks mit der linken Maustaste und ruft bei einem vorliegenden Ereignis die Methode *mouseClicked()* auf, in welcher mittels der Methode *getClickCount()* überprüft wird, ob ein doppelter Click vorliegt.

Die nachstehenden Zeilen veranschaulichen dies:

```
list.addMouseListener( new MouseListener() {
                        public void mouseClicked( MouseEvent e ) {
                                if ( e.getClickCount() == 2 ) {
                                        .................
                                }
                        }
}
```

Existiert solch ein Ereignis, so wird ein neues Objekt vom Typ *ResultSetListModel* für die Speicherung der Datensätze der nächst tieferen Ebene erstellt. Mit der Erstellung dieses Objekts werden alle für diese Ebene relevanten Daten aus der Datenbank gelesen und im Objekt gespeichert. Anschliessend wird mit der Methode *getSize()* der Klasse *ResultSetListModel* überprüft, ob dieses Objekt Datensätze enthält. Trifft dies zu, so wird das Panel der momentanen Ebene aus dem Inhaltsfenster entfernt und das unsichtbare, sich im Hintergrund befindende Panel mittels der Methode *setVisible()* sichtbar gemacht. Des Weiteren wird das neu erstellte Objekt vom Typ *ResultSetListModel* mit der Methode *setResultSetListModel()* als nun gültiger Ergebnisdatensatz erklärt und anschliessend das Inhaltsfenster aktualisiert. Zuletzt wird sowohl die *MenuBar* als auch die *ToolBar* in der Möglichkeit der Aktivierung ihrer Schaltflächen entsprechend geändert.

Folgende Zeilen, die der Klasse *SwitchListModel* entnommen sind, schildern den Wechsel von Switch-Ebene nach Cabinet-Ebene:

```
if ( cabinetListModel.getSize() > 0 ) {
        gui.removeComponent( "panelSwitch" );
        gui.getPanelCabinet().setVisible( true );
        gui.getPanelCabinet().setResultSetListModel( cabinetListModel );
        gui.setCurrentPanel( "panelCabinet" );
        gui.getContentPane().add(gui.getPanelCabinet(), BorderLayout.CENTER);
        gui.getContentPane().repaint();
        Gui.itemUpperLevel.setEnabled(true);
        Gui.itemAddSwitch.setEnabled(false);
        Gui.itemDelSwitch.setEnabled(false);
        Gui.itemAddCabinet.setEnabled(true);
        Gui.itemDelCabinet.setEnabled(true);
        Gui.itemAddShelf.setEnabled(false);
        Gui.itemDelShelf.setEnabled(false);
        Gui.itemAddCard.setEnabled(false);
        Gui.itemDelCard.setEnabled(false);
        Gui.buttonToolBarUpperLevel.setEnabled(true);
        Gui.buttonToolBarAddSwitch.setEnabled(false);
        Gui.buttonToolBarDelSwitch.setEnabled(false);
        Gui.buttonToolBarAddCabinet.setEnabled(true);
        Gui.buttonToolBarDelCabinet.setEnabled(true);
        Gui.buttonToolBarAddShelf.setEnabled(false);
        Gui.buttonToolBarDelShelf.setEnabled(false);
        Gui.buttonToolBarAddCard.setEnabled(false);
        Gui.buttonToolBarDelCard.setEnabled(false);
}
else {
        System.out.println( "Keine Datensätze vorhanden, Ebene wird nicht
                                gewechselt!" );
}
```

5.1.10 Wechsel in eine höhere Ebene

Für den Wechsel von einer Ebene in die nächst höhere Ebene wird eine Funktion bereitgestellt, welche über die in der *MenuBar* und *ToolBar* vorhandenen Schaltflächen ausgelöst wird. Diesen Schaltflächen ist ein *ActionListener* zugeordnet, der bei Ereignisauslösung über die diesen zugeordnete Methode *actionPerformed()* die Methode *upperLevelClick()* aufruft. In dieser Funktion wird an erster Stelle eine Fallunterscheidung gemacht und mittels der Methode *getCurrentPanel()* überprüft, auf welcher Ebene sich der Anwender befindet. Je nach Ebene wird das derzeit aktuelle Panel aus dem Inhaltsfenster entfernt, das Panel der sich darüber befindenden Ebene in den sichtbaren Zustand gebracht und dieses mit der Methode *add()* dem Inhaltsfenster hinzugefügt. Zuletzt wird über die Methode *setCurrentPanel()* das nun aktuelle Panel als gültiges Panel erklärt und die Schaltflächen der *MenuBar* als auch der *ToolBar* in ihrer Möglichkeit der Aktivierung entsprechend geändert.

Folgende Zeilen, die der Klasse *Gui* entnommen sind, schildern den Wechsel von Cabinet-Ebene nach Switch-Ebene:

```
if ( getCurrentPanel() == "panelCabinet" ) {
        removeComponent( "panelCabinet" );
        panelSwitch.setVisible( true );
        contentPane.add( panelSwitch, BorderLayout.CENTER );
        setCurrentPanel( "panelSwitch" );
        Gui.itemUpperLevel.setEnabled( false );
        Gui.itemAddSwitch.setEnabled( true );
        Gui.itemDelSwitch.setEnabled( true );
        Gui.itemAddCabinet.setEnabled( false );
        ..............
        ..............
}
else if ( getCurrentPanel() == "panelShelf" ) {
        ..............
}..............
```

5.1.11 Änderung des Look & Feels

Durch die Verwendung von *JAVA Swing* und der damit verbundenen *Model-View-Controller*-Architektur ist es möglich, während der Laufzeit der Applikation das Aussehen dieser zu ändern. Dieses kann über die im Menü *Look & Feel* bereitgestellten *RadioButtons* ausgewählt werden. Den *RadioButtons* sind wie auch all den anderen Schaltflächen *ActionListener* zugewiesen, welche über die ihnen zugeordnete Methode *actionPerformed()* die Methode *lookAndFeelClick()* aufrufen und mit dieser den entsprechenden String, der das *Look & Feel* spezifiziert, übergeben. Innerhalb der Methode *lookAndFeelClick()* wird der übergebene String geprüft und anhand von diesem eine Fallunterscheidung bezüglich der unterschiedlichen *Look & Feels* gemacht.
Die nachstehenden Zeilen erläutern dies näher:

```
if( lookAndFeel.equals( "java" ) ) {
        try {
            UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
            SwingUtilities.updateComponentTreeUI(getContentPane());
        } catch (Exception e){}
}
else if( lookAndFeel.equals( "motif" ) )
{
        try {
                .................
                .................
        }
}
else if.....
```

Die Methode *setLookAndFeel()* ermöglicht es, unabhängig von der aktuell benutzten Plattform das *Look & Feel* festzulegen. Hierfür muss dieser lediglich ein neues *Look & Feel*-Objekt übergeben werden.

5.1.12 Änderung der Sprache

Die Möglichkeit, die Sprache der Applikation während der Laufzeit zu ändern, wird durch den Einsatz der Internationalisierung realisiert. Mittels dieser werden die kultur- und sprachspezifischen Teile von den anderen Teilen der Anwendung getrennt und in Form von *Resource Bundles* in Textdateien ausgelagert. In dieser Anwendung existieren insgesamt vier Textdateien, die die Zeichenketten für die Beschriftung der Benutzeroberflächen-Elemente enthalten. Diese tragen die Namen

- *MessagesBundle.properties*
- *MessagesBundle_en_US.properties*
- *MessagesBundle_de_DE.properties*
- *MessagesBundle_fr_FR.properties*

Der Zugriff auf die in diesen Dateien vorliegenden Zeichenketten erfolgt über die Methode *getString()*, die durch die Klasse *ResourceBundle* zur Verfügung gestellt wird. Über die im Menü *Language* bereitgestellten *RadioButtons* lässt sich zwischen den Sprachen der Anwendung wechseln, denen wie auch all den anderen Schaltflächen *ActionListener* zugewiesen sind, welche über die ihnen zugeordnete Methode *actionPerformed()* die Methode *languageClick()* aufrufen und mit dieser die entsprechenden Strings, die für das Setzen der *Locale* notwendig sind, übergeben.
Innerhalb der Methode *languageClick()* wird anhand der übergebenen Strings eine neue *Locale* erzeugt und mittels dieser das neue *Resource Bundle* geladen. Im Anschluss daran müssen alle Beschriftungen der Benutzeroberflächen-Elemente neu gesetzt werden. Dies erfolgt für die *MenuBar* und *ToolBar* in dieser Methode selbst. Die Panels *PanelXXXDetails*, welche die Attribute der Facility enthalten, stellen hierfür eine Methode *redraw()* bereit, die diese Aufgabe übernimmt und durch die Methode *languageClick()* aufgerufen wird.

Der nachstehende Programmausschnitt veranschaulicht dies näher:

```
private void languageClick( String language, String country )
{
    main.currentLocale = new Locale( language,country );
    main.messages = ResourceBundle.getBundle( "MessagesBundle",
                    main.currentLocale );
    UIManager.put( "OptionPane.yesButtonText", main.getLabelText( "yes" ) );
    UIManager.put( "OptionPane.noButtonText", main.getLabelText( "no" ) );
    UIManager.put( "OptionPane.cancelButtonText", main.getLabelText( "cancel" ) );

    functionsMenu.setText( main.getLabelText( "functions" ) );
    lookAndFeelMenu.setText( main.getLabelText( "lookAndFeel" ) );
    languageMenu.setText( main.getLabelText( "language" ) );
    helpMenu.setText( main.getLabelText( "help" ) );

    itemUpperLevel.setText(main.getLabelText("upperLevel"));
    itemAddSwitch.setText(main.getLabelText("addSwitch"));
    itemDelSwitch.setText(main.getLabelText("deleteSwitch"));

    buttonToolBarUpperLevel.setText(main.getLabelText("upperLevel"));
    buttonToolBarAddSwitch.setText(main.getLabelText("addSwitch"));
    buttonToolBarDelSwitch.setText(main.getLabelText("deleteSwitch"));
    .............
    .............

    PanelSwitchDetails.redraw();
    PanelCabinetDetails.redraw();
    PanelShelfDetails.redraw();
    PanelCardDetails.redraw();
}
```

5.1.13 Drucken

Um die Facilities mitsamt ihren Attributen drucken zu können, wird die Methode *print()* der Klasse *PrintUtilities* zur Verfügung gestellt. Der Aufruf dieser Methode erfolgt über die hierfür zuständigen Schaltflächen, die in der *MenuBar* und *ToolBar* enthalten sind. Der diesen Schaltflächen zugeordnete *ActionListener* sorgt in seiner zugeordneten Methode *actionPerformed()* für den Aufruf der Methode *printConfigClick()*, in welcher wiederum die Methode *printComponent()* der Klasse *PrintUtilities* aktiviert wird und mittels dieser das aktuelle Objekt als Parameter übergeben wird. In dieser Methode wird ein Objekt vom Typ *PrintUtilities* erzeugt, auf welches schliesslich die Funktion *print()* angewendet wird.
In der Methode *print()* wird ein Objekt vom Typ *PrinterJob* erzeugt und hierfür mittels der Methode *getPrinterJob()* ein Druckauftrag bereitgestellt. Anschliessend wird über ein neu erzeugtes Objekt vom Typ *PageFormat* und der Methode *setOrientation()* das Seitenformat des Drucks definiert und der Druckauftrag mit der Methode *setPrintable()* als druckbar gesetzt. Im Anschluss daran ist der Druck startklar und kann mit der Methode *print()* der Klasse *PrinterJob* abgeschickt werden, vorher wird jedoch über die Methode *printDialog()* ein Druck-Dialogfenster geöffnet, welches dem Benutzer die Möglichkeit der Definition weiterer Einstellungen gibt.
Die folgenden Zeilen zeigen den Aufbau der Methode *print()*:

```
public void print() {
        PrinterJob printJob = PrinterJob.getPrinterJob();
        PageFormat pf = printJob.defaultPage();
        pf.setOrientation(PageFormat.LANDSCAPE);
        printJob.setPrintable(this, pf);
        try {
                if(printJob.printDialog())
                {
                        printJob.print();
                }
        }
        catch (PrinterException e)
        { }
  }
```

5.1.14 Verlassen der Anwendung

Das Beenden der Anwendung wird über die in der *MenuBar* und *ToolBar* zur Verfügung gestellten Schaltflächen ausgelöst. Der diesen Schaltflächen zugeordnete *ActionListener* sorgt daraufhin für den Aufruf der Methode *actionPerformed()*, welche wiederum die Methode *exitClick()* aktiviert. Um ein versehentliches Beenden der Anwendung zu verhindern, wird auch an dieser Stelle über die Methode *exitClick()* ein Dialogfenster geöffnet, welches den Benutzer auffordert, seine gewünschte Aktion zu bestätigen. Die Implementierung dieses Dialogs gleicht dem der Abfrage bei der Entfernung von Facilities. Wird die Aktion durch den Benutzer bestätigt, so erfolgt mittels dem Befehl *System.exit(0)* ein Beenden der Anwendung.

6 Zusammenfassung und Ausblick

Die vorliegende Arbeit beschreibt den Entwurf und die Realisierung des Captive Office Configuration Management Systems. Dabei wird durchgehend der objektorientierte Ansatz für die Softwareentwicklung verfolgt, der sich aus den drei Phasen Analyse, Entwurf und Implementierung zusammensetzt.

Zuerst werden in Form einer Analyse die Anforderungen an die Funktionalität, Benutzeroberfläche und an das System beschrieben. Die Analysephase stellt den wichtigsten Bestandteil der objektorientierten Softwareentwicklung dar, da diese als Grundlage für die weiteren Phasen dient. Basierend auf dem in dieser Phase erstellten Modell setzt die Entwurfsphase an und definiert ein Konzept für die Systemarchitektur, Benutzeroberfläche und Modellierung der Datenbank. Bei dieser Konzeption werden verschiedene Entwurfsmuster zur Unterstützung der Entwicklung des COCMS eingesetzt, die entsprechend in den theoretischen Grundlagen betrachtet wurden. Daraufhin wird in der Implementierungsphase anhand konkreter Anwendungsfälle auf die Realisierung eingegangen und die verwendeten Klassen mit ihren Funktionen vorgestellt. Mittels Interaktionsdiagrammen und Quelltextauszügen wird erläutert, wie die Anwendungsfälle in der Programmiersprache *JAVA* realisiert werden.

Insgesamt können der Systementwurf und die Implementierung als gelungen bezeichnet werden. Die wesentlichen Funktionalitäten konnten im Rahmen der Arbeit realisiert und die wichtigsten Anwendungsfälle abgedeckt werden.

Aufgrund der modularen Entwicklung der Software stellt es keinerlei Probleme dar, diese beliebig weiterzuentwickeln. Funktionen wie beispielsweise die Suche nach Facilities unter Vorgabe eines Suchbegriffs, die Implementierung einer Zählfunktion, die die Anzahl existierender Cards ermittelt oder gar eine Erweiterung der Druckfunktion können der Anwendung durch Entwicklung neuer Klassen beliebig hinzugefügt werden. Des Weiteren stellt es keine Schwierigkeit dar, die Architektur der Anwendung zu modifizieren. So ist es möglich, die Anwendung in Form eines *JAVA-Applets* über den Browser zugänglich zu machen oder über den Einsatz der Remote Method Invocation (RMI) eine höhere Sicherheit in Bezug auf den Datenbankzugriff, sowie eine Entlastung des Datenbankservers zu gewährleisten.

7 Quellenverzeichnis

[Berg98] Daniel J. Berg, J. Steven Fritzinger, *Advanced Techniques for Java Developers*, John Wiley & Sons, Inc., 1998.

[Burkhardt97] Rainer Burkhardt, *UML - Unified Modelling Language*, *Objektorientierte Modellierung für die Praxis*, Addison-Wesley, 1997.

[Buschmann98] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stahl, *Pattern-orientierte Software Architektur, Ein Pattern-System*, Addison-Wesley Longman Verlag GmbH, 1998.

[Cooper98] James W. Cooper, *The Design Patterns Java Companion*, Addison-Wesley, 1998

[Eckel98] Bruce Eckel, *Thinking in Java*, Prentice Hall PTR, Prentice-Hall, Inc. 1998.

[Eckstein98] Robert Eckstein, Marc Loy, Dave Wood, *Java Swing*, O'Reilly & Associates, Inc., 1998.

[Flanagan96] David Flanagan, *Java in a Nutshell*, O'Reilly, Köln, 1996.

[Fowler97] Martin Fowler, *Analysis Patterns, Reusable Object Models*, Addison-Wesley Longman, Inc., Massachusetts, 1997.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of reusable object-oriented Software*, Addison-Wesley Longman, Inc., Massachusetts, 1995.

[Grand98a] Mark Grand, *Patterns in Java, A Catalog of Reusable Design Patterns Illustrated with UML*, Willy Computer Publishing, 1998.

[Grand98b] Mark Grand, *Java Fundamental Language Reference*, O'Reilly, 1998.

[Jacobson92] Ivar Jacobson, *Object-oriented Software-Engineering, A use-case driven approach*, Addison-Wesley, Harlow, England, 1992.

[Kemper01] A. Kemper, A. Eickler, *Datenbanksysteme - Eine Einführung*, Oldenbourg Verlag, München, Wien, 2001.

[Krüger97] Guido Krüger, *Java 1.1 lernen, Anfangen, Anwenden, Verstehen*, Addison-Wesley Longman, Inc., 1999.

[Melton00] J. Melton, A. Eisenberg, *Understanding SQL and Java together*, Morgan Kaufmann, 2000.

[Meyer98] André Meyer, *JFC 1.1 mit Java Swing 1.0, Ein Tutorial für die Gestaltung graphischer Benutzerschnittstellen*, Addison-Wesley Longman Verlag GmbH, 1998.

[Oesterreich97] Bernd Oesterreich, *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*, 3. Auflage, Oldenbourg, 1997.

[Pooley99] Rob Pooley, Perdita Stevens, *Using UML, Software Engineering with Objects and Components*, Addison-Wesley Longman Limited, 1999.

[Topley98] Kim Topley, *Core, Java Foundation Classes*, Prentice-Hall Inc., 1998.

[Weiner98] Scott R. Weiner, Stephen Asbury, *Programming with JFC*, Wiley Computer Publishing, 1998.

Internet:

Java Online-Tutorial von SUN Microsystems Inc.,
http://www.javasoft.com

Java Beans,
http://www.javasoft.com/beans.index.html

Java Database Connectivity von SUN Microsystems Inc.,
http://docs.sun.com/products/jdbc/related.html

Java Swing,
http://www.javasoft.com/products/jfc/tsc/index.html

PostgreSQL,
http://www.postgresql.org/idocs

ANHANG A

A.1 Quellcode

A.1.1 Klasse Main:

```
import java.util.*;
import java.io.*;
import java.*;
import java.lang.*;
import java.sql.*;
import javax.swing.*;


class main
{
        public static Connection con;
        public static Gui objGui;
        public static String database = "comansys";
        public static String username = "cotewww";
        public static String password = "123cote";

        public static String language;
        public static String country;
        public static Locale currentLocale;
        public static ResourceBundle messages;

        public static String actualSwitch;
        public static String actualCabinet;
        public static String actualShelf;
        public static String actualCard;

        public static JFrame frame;

        public static void main(String args[]) throws ClassNotFoundException, Exception
        {
                frame = new JFrame();
                frame.getContentPane();

                Class.forName("org.postgresql.Driver");
                con =DriverManager.getConnection("jdbc:postgresql://131.147.37.81/" + database, username,
                                                                                                password);

                language = new String("en");
                country = new String("US");
                currentLocale = new Locale(language,country);
                messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);

                try
                {
                        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
                        SwingUtilities.updateComponentTreeUI(frame);
                } catch (Exception e){}

                objGui = new Gui();
                objGui.setBounds(10,0,1000,600);
                objGui.setVisible(true);
        }

        public static String getLabelText(String text)
        {
                return messages.getString(text);
        }
}
```

A.1.2 Klasse Gui:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.print.*;
import java.util.*;
import java.sql.SQLException;
import javax.swing.border.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.Iterator;


public class Gui extends JFrame
{
        private static int i=0;

        private PanelSwitch panelSwitch;
        private PanelCabinet panelCabinet;
        private PanelShelf panelShelf;
        private PanelCard panelCard;

        private Icon iconUpperLevel;
        public static Icon iconAddSwitch;
        public static Icon iconDelSwitch;
        public static Icon iconAddCabinet;
        public static Icon iconDelCabinet;
        public static Icon iconAddShelf;
        public static Icon iconDelShelf;
        public static Icon iconAddCard;
        public static Icon iconDelCard;

        private Icon iconPrintConfig;
        private Icon iconInfo;
        private Icon iconExit;

        private JMenuBar menuBar;

        private JMenu functionsMenu;
        private JMenu lookAndFeelMenu;
        private JMenu languageMenu;
        private JMenu helpMenu;

        public static JMenuItem itemUpperLevel;
        public static JMenuItem itemAddSwitch;
        public static JMenuItem itemDelSwitch;
        public static JMenuItem itemAddCabinet;
        public static JMenuItem itemDelCabinet;
        public static JMenuItem itemAddShelf;
        public static JMenuItem itemDelShelf;
        public static JMenuItem itemAddCard;
        public static JMenuItem itemDelCard;
        private JMenuItem itemPrintConfig;
        private JMenuItem itemExit;

        private ButtonGroup groupLookAndFeel;
        private JRadioButtonMenuItem itemJavaLookAndFeel;
        private JRadioButtonMenuItem itemMotifLookAndFeel;
        private JRadioButtonMenuItem itemWindowsStyleLookAndFeel;

        private ButtonGroup groupLanguage;
        private JRadioButtonMenuItem itemEnglish;
        private JRadioButtonMenuItem itemGerman;
        private JRadioButtonMenuItem itemFrench;

        private JMenuItem itemInfo;

        private JToolBar toolBar;
        public static JButton buttonToolBarUpperLevel;
        public static JButton buttonToolBarAddSwitch;
        public static JButton buttonToolBarDelSwitch;
        public static JButton buttonToolBarAddCabinet;
        public static JButton buttonToolBarDelCabinet;
        public static JButton buttonToolBarAddShelf;
        public static JButton buttonToolBarDelShelf;
        public static JButton buttonToolBarAddCard;
        public static JButton buttonToolBarDelCard;
        private JButton buttonToolBarPrintConfig;
        private JButton buttonToolBarInfo;
        private JButton buttonToolBarExit;

        public static ResultSetListModel switchListModel;
        public static Container contentPane;
        private String currentPanel;
```

```
public Gui()
{
        super("Captive Office Configuration Management System");
        createGUI();
        addListeners();
}


private void createGUI()
{
        UIManager.put("OptionPane.yesButtonText", main.getLabelText("yes"));
        UIManager.put("OptionPane.noButtonText", main.getLabelText("no"));
        UIManager.put("OptionPane.cancelButtonText", main.getLabelText("cancel"));

        i+=25;
        setBounds(i,i,800,600);

        contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        Border border = BorderFactory.createRaisedBevelBorder();

        iconUpperLevel = new ImageIcon("upperlevel.gif");
        iconAddSwitch = new ImageIcon("addswitch.gif");
        iconDelSwitch = new ImageIcon("delswitch.gif");
        iconAddCabinet = new ImageIcon("addcabinet.gif");
        iconDelCabinet = new ImageIcon("delcabinet.gif");
        iconAddShelf = new ImageIcon("addshelf.gif");
        iconDelShelf = new ImageIcon("delshelf.gif");
        iconAddCard = new ImageIcon("addcard.gif");
        iconDelCard = new ImageIcon("delcard.gif");
        iconPrintConfig = new ImageIcon("printconfig.gif");
        iconInfo = new ImageIcon("info.gif");
        iconExit = new ImageIcon("exit.gif");

        menuBar = new JMenuBar();

        functionsMenu = new JMenu(main.getLabelText("functions"));
        itemUpperLevel = new JMenuItem(main.getLabelText("upperLevel"), iconUpperLevel);
        itemUpperLevel.setEnabled(false);
        itemAddSwitch = new JMenuItem(main.getLabelText("addSwitch"), iconAddSwitch);
        itemDelSwitch = new JMenuItem(main.getLabelText("deleteSwitch"), iconDelSwitch);
        itemAddCabinet = new JMenuItem(main.getLabelText("addCabinet"), iconAddCabinet);
        itemAddCabinet.setEnabled(false);
        itemDelCabinet = new JMenuItem(main.getLabelText("deleteCabinet"), iconDelCabinet);
        itemDelCabinet.setEnabled(false);
        itemAddShelf = new JMenuItem(main.getLabelText("addShelf"), iconAddShelf);
        itemAddShelf.setEnabled(false);
        itemDelShelf = new JMenuItem(main.getLabelText("deleteShelf"), iconDelShelf);
        itemDelShelf.setEnabled(false);
        itemAddCard = new JMenuItem(main.getLabelText("addCard"), iconAddCard);
        itemAddCard.setEnabled(false);
        itemDelCard = new JMenuItem(main.getLabelText("deleteCard"), iconDelCard);
        itemDelCard.setEnabled(false);
        itemPrintConfig = new JMenuItem(main.getLabelText("printConfiguration"), iconPrintConfig);
        itemExit = new JMenuItem(main.getLabelText("exit"), iconExit);

        functionsMenu.add(itemUpperLevel);
        functionsMenu.addSeparator();
        functionsMenu.add(itemAddSwitch);
        functionsMenu.add(itemDelSwitch);
        functionsMenu.add(itemAddCabinet);
        functionsMenu.add(itemDelCabinet);
        functionsMenu.add(itemAddShelf);
        functionsMenu.add(itemDelShelf);
        functionsMenu.add(itemAddCard);
        functionsMenu.add(itemDelCard);
        functionsMenu.addSeparator();
        functionsMenu.add(itemPrintConfig);
        functionsMenu.addSeparator();
        functionsMenu.add(itemExit);

        lookAndFeelMenu = new JMenu(main.getLabelText("lookAndFeel"));
        groupLookAndFeel = new ButtonGroup();
        itemJavaLookAndFeel = new JRadioButtonMenuItem(main.getLabelText("javaLookAndFeel"));
        itemMotifLookAndFeel = new JRadioButtonMenuItem(main.getLabelText("motifLookAndFeel"));
        itemWindowsStyleLookAndFeel =
                new JRadioButtonMenuItem(main.getLabelText("windowsStyleLookAndFeel"));

        lookAndFeelMenu.add(itemJavaLookAndFeel);
        lookAndFeelMenu.add(itemMotifLookAndFeel);
        lookAndFeelMenu.add(itemWindowsStyleLookAndFeel);
        groupLookAndFeel.add(itemJavaLookAndFeel);
        groupLookAndFeel.add(itemMotifLookAndFeel);
        groupLookAndFeel.add(itemWindowsStyleLookAndFeel);
```

```
languageMenu = new JMenu(main.getLabelText("language"));
groupLanguage = new ButtonGroup();
itemEnglish = new JRadioButtonMenuItem(main.getLabelText("english"));
itemGerman = new JRadioButtonMenuItem(main.getLabelText("german"));
itemFrench = new JRadioButtonMenuItem(main.getLabelText("french"));
languageMenu.add(itemEnglish);
languageMenu.add(itemGerman);
languageMenu.add(itemFrench);
groupLanguage.add(itemEnglish);
groupLanguage.add(itemGerman);
groupLanguage.add(itemFrench);
itemEnglish.setSelected(true);

helpMenu = new JMenu(main.getLabelText("help"));
itemInfo = new JMenuItem(main.getLabelText("info"), iconInfo);
helpMenu.add(itemInfo);

menuBar.add(functionsMenu);
menuBar.add(lookAndFeelMenu);
menuBar.add(languageMenu);
menuBar.add(helpMenu);
setJMenuBar(menuBar);

JToolBar toolBar = new JToolBar();
toolBar.setFloatable(false);

buttonToolBarUpperLevel = new JButton(main.getLabelText("upperLevel"), iconUpperLevel);
buttonToolBarUpperLevel.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarUpperLevel.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarUpperLevel.setEnabled(false);
buttonToolBarAddSwitch = new JButton(main.getLabelText("addSwitch"), iconAddSwitch);
buttonToolBarAddSwitch.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarAddSwitch.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarDelSwitch = new JButton(main.getLabelText("deleteSwitch"), iconDelSwitch);
buttonToolBarDelSwitch.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarDelSwitch.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarAddCabinet = new JButton(main.getLabelText("addCabinet"), iconAddCabinet);
buttonToolBarAddCabinet.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarAddCabinet.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarAddCabinet.setEnabled(false);
buttonToolBarDelCabinet = new JButton(main.getLabelText("deleteCabinet"), iconDelCabinet);
buttonToolBarDelCabinet.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarDelCabinet.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarDelCabinet.setEnabled(false);
buttonToolBarAddShelf = new JButton(main.getLabelText("addShelf"), iconAddShelf);
buttonToolBarAddShelf.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarAddShelf.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarAddShelf.setEnabled(false);
buttonToolBarDelShelf = new JButton(main.getLabelText("deleteShelf"), iconDelShelf);
buttonToolBarDelShelf.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarDelShelf.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarDelShelf.setEnabled(false);
buttonToolBarAddCard = new JButton(main.getLabelText("addCard"), iconAddCard);
buttonToolBarAddCard.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarAddCard.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarAddCard.setEnabled(false);
buttonToolBarDelCard = new JButton(main.getLabelText("deleteCard"), iconDelCard);
buttonToolBarDelCard.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarDelCard.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarDelCard.setEnabled(false);
buttonToolBarPrintConfig = new JButton(main.getLabelText("printConfiguration"),
                                                              iconPrintConfig);
buttonToolBarPrintConfig.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarPrintConfig.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarInfo = new JButton(main.getLabelText("info"), iconInfo);
buttonToolBarInfo.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarInfo.setHorizontalTextPosition(SwingConstants.CENTER);
buttonToolBarExit = new JButton(main.getLabelText("exit"), iconExit);
buttonToolBarExit.setVerticalTextPosition(SwingConstants.BOTTOM);
buttonToolBarExit.setHorizontalTextPosition(SwingConstants.CENTER);

toolBar.add(buttonToolBarUpperLevel);
toolBar.add(buttonToolBarAddSwitch);
toolBar.add(buttonToolBarDelSwitch);
toolBar.add(buttonToolBarAddCabinet);
toolBar.add(buttonToolBarDelCabinet);
toolBar.add(buttonToolBarAddShelf);
toolBar.add(buttonToolBarDelShelf);
toolBar.add(buttonToolBarAddCard);
toolBar.add(buttonToolBarDelCard);
toolBar.add(buttonToolBarPrintConfig);
toolBar.add(buttonToolBarInfo);
toolBar.addSeparator();
toolBar.add(buttonToolBarExit);
toolBar.putClientProperty("JToolBar.isRollover", Boolean.TRUE);

contentPane.add(toolBar, BorderLayout.NORTH);
```

```
		try
		{
			switchListModel = new ResultSetListModel( "switch", "pk_switch");
		}
		catch ( SQLException e ) {
			e.printStackTrace();
		}

		panelCabinet = new PanelCabinet( this );
		panelCabinet.setBorder( BorderFactory.createTitledBorder( "Available Cabinets" ));
		panelCabinet.setName( "panelCabinet" );
		panelShelf = new PanelShelf( this );
		panelShelf.setBorder( BorderFactory.createTitledBorder( "Available Shelves" ));
		panelShelf.setName( "panelShelf" );
		panelCard = new PanelCard( this );
		panelCard.setBorder( BorderFactory.createTitledBorder( "Available Cards" ));
		panelCard.setName( "panelCard" );
		panelSwitch = new PanelSwitch( this );
		panelSwitch.setResultSetListModel( switchListModel );
		panelSwitch.setBorder(BorderFactory.createTitledBorder( "Available Switches"));
		panelSwitch.setName( "panelSwitch" );
		contentPane.add(panelSwitch, BorderLayout.CENTER);
}

private void addListeners()
{
		this.addWindowListener(new WindowAdapter()
		{
			public void windowClosing(WindowEvent e)
			{
				i-=25;
				setVisible(false);
				dispose();
			}
		});

		itemUpperLevel.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent theEvent)
			{
				upperLevelClick();
			}
		});

		itemAddSwitch.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent theEvent)
			{
				addSwitchClick();
			}
		});

		itemDelSwitch.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent theEvent)
			{
				delSwitchClick();
			}
		});

		itemAddCabinet.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent theEvent)
			{
				addCabinetClick();
			}
		});

		itemDelCabinet.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent theEvent)
			{
				delCabinetClick();
			}
		});

		itemAddShelf.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent theEvent)
			{
				addShelfClick();
			}
		});
```

```
itemDelShelf.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                delShelfClick();
        }
});

itemAddCard.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                addCardClick();
        }
});

itemDelCard.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                delCardClick();
        }
});

itemPrintConfig.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                printConfigClick();
        }
});

itemExit.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                exitClick();
        }
});

itemJavaLookAndFeel.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                lookAndFeelClick("java");
        }
});

itemMotifLookAndFeel.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                lookAndFeelClick("motif");
        }
});

itemWindowsStyleLookAndFeel.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                lookAndFeelClick("windows");
        }
});

itemEnglish.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                languageClick("en", "US");
        }
});

itemGerman.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                languageClick("de", "DE");
        }
});

itemFrench.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                languageClick("fr", "FR");
        }
});
```

```
itemInfo.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                infoClick();
        }
});

buttonToolBarUpperLevel.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                upperLevelClick();
        }
});

buttonToolBarAddSwitch.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                addSwitchClick();
        }
});

buttonToolBarDelSwitch.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                delSwitchClick();
        }
});

buttonToolBarAddCabinet.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                addCabinetClick();
        }
});

buttonToolBarDelCabinet.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                delCabinetClick();
        }
});

buttonToolBarAddShelf.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                addShelfClick();
        }
});

buttonToolBarDelShelf.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                delShelfClick();
        }
});

buttonToolBarAddCard.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                addCardClick();
        }
});

buttonToolBarDelCard.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                delCardClick();
        }
});

buttonToolBarPrintConfig.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent theEvent)
        {
                printConfigClick();
        }
});
```

```
        buttonToolBarInfo.addActionListener(new ActionListener()
        {
                public void actionPerformed(ActionEvent theEvent)
                {
                        infoClick();
                }
        });

        buttonToolBarExit.addActionListener(new ActionListener()
        {
                public void actionPerformed(ActionEvent theEvent)
                {
                        exitClick();
                }
        });
}

private void upperLevelClick()
{
        if ( getCurrentPanel() == "panelCabinet" ) {
                removeComponent( "panelCabinet" );
                panelSwitch.setVisible(true);
                contentPane.add( panelSwitch, BorderLayout.CENTER );

                setCurrentPanel( "panelSwitch" );

                Gui.itemUpperLevel.setEnabled(false);
                Gui.itemAddSwitch.setEnabled(true);
                Gui.itemDelSwitch.setEnabled(true);
                Gui.itemAddCabinet.setEnabled(false);
                Gui.itemDelCabinet.setEnabled(false);
                Gui.itemAddShelf.setEnabled(false);
                Gui.itemDelShelf.setEnabled(false);
                Gui.itemAddCard.setEnabled(false);
                Gui.itemDelCard.setEnabled(false);
                Gui.buttonToolBarUpperLevel.setEnabled(false);
                Gui.buttonToolBarAddSwitch.setEnabled(true);
                Gui.buttonToolBarDelSwitch.setEnabled(true);
                Gui.buttonToolBarAddCabinet.setEnabled(false);
                Gui.buttonToolBarDelCabinet.setEnabled(false);
                Gui.buttonToolBarAddShelf.setEnabled(false);
                Gui.buttonToolBarDelShelf.setEnabled(false);
                Gui.buttonToolBarAddCard.setEnabled(false);
                Gui.buttonToolBarDelCard.setEnabled(false);
        }
        else if ( getCurrentPanel() == "panelShelf" ) {
                removeComponent( "panelShelf" );
                panelCabinet.setVisible(true);
                contentPane.add( panelCabinet, BorderLayout.CENTER );

                setCurrentPanel( "panelCabinet" );

                Gui.itemUpperLevel.setEnabled(true);
                Gui.itemAddSwitch.setEnabled(false);
                Gui.itemDelSwitch.setEnabled(false);
                Gui.itemAddCabinet.setEnabled(true);
                Gui.itemDelCabinet.setEnabled(true);
                Gui.itemAddShelf.setEnabled(false);
                Gui.itemDelShelf.setEnabled(false);
                Gui.itemAddCard.setEnabled(false);
                Gui.itemDelCard.setEnabled(false);
                Gui.buttonToolBarUpperLevel.setEnabled(true);
                Gui.buttonToolBarAddSwitch.setEnabled(false);
                Gui.buttonToolBarDelSwitch.setEnabled(false);
                Gui.buttonToolBarAddCabinet.setEnabled(true);
                Gui.buttonToolBarDelCabinet.setEnabled(true);
                Gui.buttonToolBarAddShelf.setEnabled(false);
                Gui.buttonToolBarDelShelf.setEnabled(false);
                Gui.buttonToolBarAddCard.setEnabled(false);
                Gui.buttonToolBarDelCard.setEnabled(false);
        }
        else if ( getCurrentPanel() == "panelCard") {
                removeComponent( "panelCard" );
                panelShelf.setVisible(true);
                contentPane.add( panelShelf, BorderLayout.CENTER );

                setCurrentPanel( "panelShelf" );

                Gui.itemUpperLevel.setEnabled(true);
                Gui.itemAddSwitch.setEnabled(false);
                Gui.itemDelSwitch.setEnabled(false);
                Gui.itemAddCabinet.setEnabled(false);
                Gui.itemDelCabinet.setEnabled(false);
                Gui.itemAddShelf.setEnabled(true);
                Gui.itemDelShelf.setEnabled(true);
                Gui.itemAddCard.setEnabled(false);
                Gui.itemDelCard.setEnabled(false);
```

```
                Gui.buttonToolBarUpperLevel.setEnabled(true);
                Gui.buttonToolBarAddSwitch.setEnabled(false);
                Gui.buttonToolBarDelSwitch.setEnabled(false);
                Gui.buttonToolBarAddCabinet.setEnabled(false);
                Gui.buttonToolBarDelCabinet.setEnabled(false);
                Gui.buttonToolBarAddShelf.setEnabled(true);
                Gui.buttonToolBarDelShelf.setEnabled(true);
                Gui.buttonToolBarAddCard.setEnabled(false);
                Gui.buttonToolBarDelCard.setEnabled(false);
        }
}

private void addSwitchClick()
{
        String s = JOptionPane.showInputDialog(this,
                                                main.getLabelText("addSwitchDialog"),
                                                main.getLabelText("addSwitch") ,
                                                JOptionPane.QUESTION_MESSAGE);

        ((ResultSetListModel)switchListModel).addElement(s);
}

private void delSwitchClick()
{
        try
        {
                Toolkit.getDefaultToolkit().beep();
                int reply = JOptionPane.showConfirmDialog(this,
                                                main.getLabelText("deleteSwitchDialog"),
                                                main.getLabelText("deleteSwitch") ,
                                                JOptionPane.YES_NO_OPTION,
                                                JOptionPane.QUESTION_MESSAGE);
                if (reply == JOptionPane.YES_OPTION)
                {
                        int selected = ((ResultSetListModel)switchListModel).getIndex();
                        ((ResultSetListModel)switchListModel).remove(selected);
                }
        } catch (Exception e) {}
}

private void addCabinetClick()
{
        String s = JOptionPane.showInputDialog( this, main.getLabelText("addCabinetDialog"),
                        main.getLabelText("addCabinet"), JOptionPane.QUESTION_MESSAGE);
        ((PanelCabinet)contentPane.getComponentAt( contentPane.getSize().width/2,
                        contentPane.getSize().height/2)).getPanelCabinetList().
                        getResultSetListModel().addElement(s);
}

private void delCabinetClick()
{
        try
        {
                Toolkit.getDefaultToolkit().beep();
                int reply = JOptionPane.showConfirmDialog(this,
                                                main.getLabelText("deleteCabinetDialog"),
                                                main.getLabelText("deleteCabinet") ,
                                                JOptionPane.YES_NO_OPTION,
                                                JOptionPane.QUESTION_MESSAGE);
                if (reply == JOptionPane.YES_OPTION)
                {
                        int selected = ((PanelCabinet)contentPane.getComponentAt(
                                contentPane.getSize().width/2, contentPane.getSize().height/2
                                )).getPanelCabinetList().getResultSetListModel().getIndex();
                        ((PanelCabinet)contentPane.getComponentAt( contentPane.getSize().width/2,
                                contentPane.getSize().height/2 )).getPanelCabinetList().
                                getResultSetListModel().remove(selected);
                }
        } catch (Exception e) {}
}

public static String addFirstCabinet()
{
        String s = JOptionPane.showInputDialog( contentPane,
                main.getLabelText("addFirstCabinetDialog"), main.getLabelText("addFirstCabinet"),
                JOptionPane.QUESTION_MESSAGE);
        return s;
}

public static String addFirstShelf()
{
        String s = JOptionPane.showInputDialog( contentPane,
                main.getLabelText("addFirstShelfDialog"), main.getLabelText("addFirstShelf"),
                JOptionPane.QUESTION_MESSAGE);
        return s;
}
```

```
public static String addFirstCard()
{
        String s = JOptionPane.showInputDialog( contentPane,
                main.getLabelText("addFirstCardDialog"), main.getLabelText("addFirstCard"),
                JOptionPane.QUESTION_MESSAGE);
        return s;
}

private void addShelfClick()
{
        String s = JOptionPane.showInputDialog(this,
                                                main.getLabelText("addShelfDialog"),
                                                main.getLabelText("addShelf") ,
                                                JOptionPane.QUESTION_MESSAGE);
        ((PanelShelf)contentPane.getComponentAt( contentPane.getSize().width/2,
                contentPane.getSize().height/2 )).getPanelShelfList().
                getResultSetListModel().addElement(s);
}

private void delShelfClick()
{
        try
        {
                Toolkit.getDefaultToolkit().beep();
                int reply = JOptionPane.showConfirmDialog(this,
                                                main.getLabelText("deleteShelfDialog"),
                                                main.getLabelText("deleteShelf") ,
                                                JOptionPane.YES_NO_OPTION,
                                                JOptionPane.QUESTION_MESSAGE);
                if (reply == JOptionPane.YES_OPTION)
                {
                        int selected = ((PanelShelf)contentPane.getComponentAt(
                                contentPane.getSize().width/2,
                                contentPane.getSize().height/2)).getPanelShelfList().
                                getResultSetListModel().getIndex();
                        ((PanelShelf)contentPane.getComponentAt( contentPane.getSize().width/2,
                                contentPane.getSize().height/2 )).
                                getPanelShelfList().getResultSetListModel().remove(selected);
                }
        } catch (Exception e) {}
}

private void addCardClick()
{
        String s = JOptionPane.showInputDialog(this,
                                                main.getLabelText("addCardDialog"),
                                                main.getLabelText("addCard") ,
                                                JOptionPane.QUESTION_MESSAGE);
        ((PanelCard)contentPane.getComponentAt( contentPane.getSize().width/2,
                contentPane.getSize().height/2 )).getPanelCardList().
                getResultSetListModel().addElement(s);
}

private void delCardClick()
{
        try
        {
                Toolkit.getDefaultToolkit().beep();
                int reply = JOptionPane.showConfirmDialog(this,
                                                main.getLabelText("deleteCardDialog"),
                                                main.getLabelText("deleteCard") ,
                                                JOptionPane.YES_NO_OPTION,
                                                JOptionPane.QUESTION_MESSAGE);
                if (reply == JOptionPane.YES_OPTION)
                {
                        int selected = ((PanelCard)contentPane.getComponentAt(
                                contentPane.getSize().width/2,
                                contentPane.getSize().height/2 )).
                                getPanelCardList().getResultSetListModel().getIndex();
                        ((PanelCard)contentPane.getComponentAt( contentPane.getSize().width/2,
                                contentPane.getSize().height/2 )).
                                getPanelCardList().getResultSetListModel().remove(selected);

                }
        } catch (Exception e) {}
}

private void printConfigClick()
{
        PrintUtilities.printComponent(this);
}
```

```
private void infoClick()
{
        String infoClickTitle = "Info";
        String infoClickMessage = "Captive Office Configuration Management System.\n\n"+
        "Copyright © 2002 Nortel Networks Germany.\n\n"+"All rights reserved.\n\n";
        JOptionPane infoPane = new JOptionPane(infoClickMessage, JOptionPane.INFORMATION_MESSAGE);
        JDialog dialog = infoPane.createDialog(new JButton(), infoClickTitle);
        dialog.show();
}

private void exitClick()
{
        try
        {
                Toolkit.getDefaultToolkit().beep();
                int reply = JOptionPane.showConfirmDialog(this,
                                                main.getLabelText("exitDialog"),
                                                main.getLabelText("exit") ,
                                                JOptionPane.YES_NO_OPTION,
                                                JOptionPane.QUESTION_MESSAGE);
                if (reply == JOptionPane.YES_OPTION)
                {
                        this.setVisible(false);
                        this.dispose();
                        System.exit(0);
                }
        } catch (Exception e) {}
}

private void lookAndFeelClick(String lookAndFeel)
{
        if(lookAndFeel.equals("java"))
        {
                try
                {
                        UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
                        SwingUtilities.updateComponentTreeUI(getContentPane());
                } catch (Exception e){}
        }

        else if(lookAndFeel.equals("motif"))
        {
                try
                {
                        UIManager.setLookAndFeel
                                ("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
                        SwingUtilities.updateComponentTreeUI(getContentPane());
                } catch (Exception e){}
        }

        else if(lookAndFeel.equals("windows"))
        {
                try
                {
                        UIManager.setLookAndFeel
                                ("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
                        SwingUtilities.updateComponentTreeUI(getContentPane());
                } catch (Exception e){}
        }
}

private void languageClick(String language, String country)
{
        main.currentLocale = new Locale(language,country);
        main.messages = ResourceBundle.getBundle("MessagesBundle", main.currentLocale);

        UIManager.put("OptionPane.yesButtonText", main.getLabelText("yes"));
        UIManager.put("OptionPane.noButtonText", main.getLabelText("no"));
        UIManager.put("OptionPane.cancelButtonText", main.getLabelText("cancel"));

        functionsMenu.setText(main.getLabelText("functions"));
        lookAndFeelMenu.setText(main.getLabelText("lookAndFeel"));
        languageMenu.setText(main.getLabelText("language"));
        helpMenu.setText(main.getLabelText("help"));

        itemUpperLevel.setText(main.getLabelText("upperLevel"));
        itemAddSwitch.setText(main.getLabelText("addSwitch"));
        itemDelSwitch.setText(main.getLabelText("deleteSwitch"));
        itemAddCabinet.setText(main.getLabelText("addCabinet"));
        itemDelCabinet.setText(main.getLabelText("deleteCabinet"));
        itemAddShelf.setText(main.getLabelText("addShelf"));
        itemDelShelf.setText(main.getLabelText("deleteShelf"));
        itemAddCard.setText(main.getLabelText("addCard"));
        itemDelCard.setText(main.getLabelText("deleteCard"));
        itemPrintConfig.setText(main.getLabelText("printConfiguration"));
        itemExit.setText(main.getLabelText("exit"));
```

```
                itemJavaLookAndFeel.setText(main.getLabelText("javaLookAndFeel"));
                itemMotifLookAndFeel.setText(main.getLabelText("motifLookAndFeel"));
                itemWindowsStyleLookAndFeel.setText(main.getLabelText("windowsStyleLookAndFeel"));

                itemEnglish.setText(main.getLabelText("english"));
                itemGerman.setText(main.getLabelText("german"));
                itemFrench.setText(main.getLabelText("french"));

                itemInfo.setText(main.getLabelText("info"));

                buttonToolBarUpperLevel.setText(main.getLabelText("upperLevel"));
                buttonToolBarAddSwitch.setText(main.getLabelText("addSwitch"));
                buttonToolBarDelSwitch.setText(main.getLabelText("deleteSwitch"));
                buttonToolBarAddCabinet.setText(main.getLabelText("addCabinet"));
                buttonToolBarDelCabinet.setText(main.getLabelText("deleteCabinet"));
                buttonToolBarAddShelf.setText(main.getLabelText("addShelf"));
                buttonToolBarDelShelf.setText(main.getLabelText("deleteShelf"));
                buttonToolBarAddCard.setText(main.getLabelText("addCard"));
                buttonToolBarDelCard.setText(main.getLabelText("deleteCard"));
                buttonToolBarPrintConfig.setText(main.getLabelText("printConfiguration"));
                buttonToolBarInfo.setText(main.getLabelText("info"));
                buttonToolBarExit.setText(main.getLabelText("exit"));

                PanelSwitchDetails.redraw();
                PanelCabinetDetails.redraw();
                PanelShelfDetails.redraw();
                PanelCardDetails.redraw();
        }

        public String getLabelText(String text)
        {
                return main.messages.getString(text);
        }


        public void setCurrentPanel( String panel ) { this.currentPanel = panel; }
        public String getCurrentPanel() { return currentPanel; }

        public PanelSwitch getPanelSwitch() { return panelSwitch; }
        public PanelCabinet getPanelCabinet() { return panelCabinet; }
        public PanelShelf getPanelShelf() { return panelShelf; }
        public PanelCard getPanelCard() { return panelCard; }

        public void removeComponent( String name ) {

                Component[] components = contentPane.getComponents();
                for ( int i = 0; i < components.length; ++i ) {
                        try {
                                if ( components[i].getName().equals( name ) ) {
                                        components[i].setVisible( false );
                                        contentPane.remove( components[i] );
                                        break;
                                }
                        }
                        catch ( NullPointerException ee ) {
                                System.out.println( "Objektname war NULL." );
                        }
                }

        }
}
```

A.1.3 Klasse NameAndPictureListCellRenderer:

```
import java.awt.Component;
import java.awt.Color;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.ListCellRenderer;
import javax.swing.border.Border;
import javax.swing.BorderFactory;


class NameAndPictureListCellRenderer extends JLabel implements ListCellRenderer
{
        private Border lineBorder = BorderFactory.createLineBorder(Color.red, 2);
        private Border emptyBorder = BorderFactory.createEmptyBorder(2,2,2,2);

        public NameAndPictureListCellRenderer()
        {
                setOpaque(true);
        }

        public Component getListCellRendererComponent( JList list, Object value, int index,
                                                        boolean  isSelected, boolean cellHasFocus)
        {
                ResultSetListModel model = (ResultSetListModel)list.getModel();

                setText( model.getElementAtForList( index ) );

                if( isSelected ) {
                        setForeground( list.getSelectionForeground() );
                        setBackground( list.getSelectionBackground() );
                }
                else {
                        setForeground( list.getForeground() );
                        setBackground( list.getBackground() );
                }

                if( cellHasFocus )
                        setBorder( lineBorder );
                else
                        setBorder( emptyBorder );

                return this;

        }

}
```

A.1.4 Klasse ResultSetListModel:

```
import java.util.HashMap;
import java.util.Iterator;
import java.net.URL;
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;
import java.sql.Statement;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.AbstractListModel;


public class ResultSetListModel extends AbstractListModel
{
        private int index;
        private int size;
        private String tableName;
        private String pkFieldName;
        private String fkFieldName;
        private String fkFieldValue;
        private PropertyChangeSupport pcs;
        private ResultSet rs;
        private Statement sql;

        public ResultSetListModel( String tableName, String pkFieldName ) throws SQLException
        {
                super();
                this.tableName = tableName;
                this.pkFieldName = pkFieldName;
                pcs = new PropertyChangeSupport(this);
                Statement stmt = main.con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                                ResultSet.CONCUR_UPDATABLE );
                rs = stmt.executeQuery( "SELECT * FROM " + tableName + " ORDER BY switchname" );
                rs.last();
                this.size = rs.getRow();
                rs.first();
                index = 0;
        }

        public ResultSetListModel( String tableName, String pkFieldName, String fkFieldName, String
                                                        fkFieldValue ) throws SQLException
        {
                super();
                this.tableName = tableName;
                this.pkFieldName = pkFieldName;
                this.fkFieldName = fkFieldName;
                this.fkFieldValue = fkFieldValue;
                pcs = new PropertyChangeSupport(this);
                Statement stmt = main.con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                                ResultSet.CONCUR_UPDATABLE);

                try {
                        if (tableName == "cabinet")
                                rs = stmt.executeQuery( "SELECT * FROM " + tableName + " WHERE " +
                                        fkFieldName + " = '" + fkFieldValue + "' ORDER BY cabinetname");
                        else if (tableName == "shelf")
                                rs = stmt.executeQuery( "SELECT * FROM " + tableName + " WHERE " +
                                        fkFieldName + " = '" + fkFieldValue + "' ORDER BY shelfname");
                        else if (tableName == "card")
                                rs = stmt.executeQuery( "SELECT * FROM " + tableName + " WHERE " +
                                        fkFieldName + " = '" + fkFieldValue + "' ORDER BY cardpeccode");

                        if ( rs.next() )
                        {
                                rs.last();
                                this.size = rs.getRow();
                                rs.first();
                                index = 0;
                        }
                        else
                        {
                                String s = null;
                                if ( tableName == "cabinet" ) {
                                        s = Gui.addFirstCabinet();
                                }
                                else if ( tableName == "shelf" ) {
                                        s = Gui.addFirstShelf();
                                }
                                else if ( tableName == "card" ) {
                                        s = Gui.addFirstCard();
                                }
```

```
                        if ( s != null )
                        {
                                if ( s.length() > 0 )
                                        addElement( s );
                        }
                        else
                        {
                                size = 0;
                                index = -1;
                        }
                }
        }
        catch ( SQLException e )
        {
                e.printStackTrace();
        }
}

public boolean setIndex( int index )
{
        if ( index <=0 )
        {
                index = 0;
        }

        int indexOld = this.index;
        this.index = index;

        pcs.firePropertyChange( "index", indexOld, index );

        try
        {
                rs.absolute(index+1);
        }
        catch ( SQLException e )
        {
                e.printStackTrace();
                return false;
        }

        if ( indexOld != index )
        {
                pcs.firePropertyChange( "index", indexOld, index );
        }

        return true;
}

public int getIndex()
{
        return index;
}

public void setSize( int size )
{
        if ( size != this.size )
        {
                this.size = size;
                pcs.firePropertyChange( "size", this.size, size );
        }
}

public Object getElementAt( int index )
{
        index++;
        Object o = null;
        try
        {
                int row = rs.getRow();
                rs.absolute( index );
                o = rs.getObject( pkFieldName );
                rs.absolute( row );
        }
        catch ( SQLException e )
        {
                e.printStackTrace();
                return null;
        }

        return o;

}
```

```
public String getElementAtForList( int index )
{
        index++;
        String o = null;
        try
        {
                int row = rs.getRow();
                rs.absolute( index );

                if (tableName == "switch")
                        o = rs.getString("switchname");
                else if (tableName == "cabinet")
                        o = rs.getString("cabinetname");
                else if (tableName == "shelf")
                        o = rs.getString("shelfname");
                else if (tableName == "card")
                        o = rs.getString("cardpeccode");

                rs.absolute( row );
        }
        catch ( SQLException e )
        {
                e.printStackTrace();
                return null;
        }
        return o;
}

public String getString(String columnName)
{
        try
        {
                String getStringResult = rs.getString(columnName);
                return getStringResult;
        }
        catch ( SQLException e )
        {
                e.printStackTrace();
                return null;
        }
        catch ( Exception e )
        {
                return null;
        }
}

public int getSize() { return size; }

public boolean addElement( Object elementName )
{
       try
       {
               Statement sql = main.con.createStatement();

                if (tableName == "switch")
                {
                        String sqlText = "INSERT INTO " + tableName + " ( switchname )" + "VALUES
                                                          ('" + elementName.toString() + "')";
                        sql.executeUpdate(sqlText);
                }
                else if (tableName == "cabinet")
                {
                        String sqlText = "INSERT INTO " + tableName + " ( " + fkFieldName + ",
                                        cabinetname )" + "VALUES ('" + fkFieldValue + "', '" +
                                        elementName.toString() + "')";
                        sql.executeUpdate(sqlText);
                }
                else if (tableName == "shelf")
                {
                        String sqlText = "INSERT INTO " + tableName + " ( " + fkFieldName + ",
                                        shelfname )" + "VALUES ('" + fkFieldValue + "', '" +
                                        elementName.toString() + "')";
                        sql.executeUpdate(sqlText);
                }

                else if (tableName == "card")
                {
                        String sqlText = "INSERT INTO " + tableName + " ( " + fkFieldName + ",
                                        cardpeccode )" + "VALUES ('" + fkFieldValue + "', '" +
                                        elementName.toString() + "')";
                        sql.executeUpdate(sqlText);
                }

                Statement stmt = main.con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                        ResultSet.CONCUR_UPDATABLE );
```

```
                if (tableName == "switch")
                        rs = stmt.executeQuery( "SELECT * FROM " + tableName + " ORDER BY
                                                                                switchname" );
                else if (tableName == "cabinet")
                        rs = stmt.executeQuery( "SELECT * FROM " + tableName + " WHERE " +
                                fkFieldName + " = '" + fkFieldValue + "' ORDER BY cabinetname" );
                else if (tableName == "shelf")
                        rs = stmt.executeQuery( "SELECT * FROM " + tableName + " WHERE " +
                                fkFieldName + " = '" + fkFieldValue + "' ORDER BY shelfname" );
                else if (tableName == "card")
                        rs = stmt.executeQuery( "SELECT * FROM " + tableName + " WHERE " +
                                fkFieldName + " = '" + fkFieldValue + "' ORDER BY cardpeccode" );

                rs.last();
                setSize( rs.getRow() );

                setIndex( rs.getRow() -1 );

                fireIntervalAdded( this, index, index+1 );
        }
        catch ( SQLException e )
        {
                e.printStackTrace();
                return false;
        }

        return true;
}

public boolean remove( int index )
{
         try
         {
                if ( this.index != index ) rs.absolute( index );

                Statement sql = main.con.createStatement();

                if (tableName == "switch")
                {
                        String sqlText = "DELETE FROM " + tableName + " WHERE " + pkFieldName + "
                                                = '" + (String)rs.getString("pk_switch") + "'";
                        sql.executeUpdate(sqlText);
                }
                else
                {
                        String sqlText = "DELETE FROM " + tableName + " WHERE " + pkFieldName + "
                                                = '" + (String)rs.getString(pkFieldName) + "'";
                        sql.executeUpdate(sqlText);
                }

                Statement stmt = main.con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                                ResultSet.CONCUR_UPDATABLE );

                if (tableName == "switch")
                {
                rs = stmt.executeQuery( "SELECT * FROM " + tableName + " ORDER BY " +
                                                                                pkFieldName );
         }
         else
         {
                rs = stmt.executeQuery( "SELECT * FROM " + tableName + " WHERE " + fkFieldName + "
                                        = '" + fkFieldValue + "' ORDER BY " + pkFieldName );
         }

                rs.last();
                int i = getSize();
                setSize( --i );

                setIndex( rs.getRow() -1 );

                fireIntervalRemoved( this, index, index-1);
         }
         catch ( SQLException e )
         {
                e.printStackTrace();
                return false;
         }

         return true;

}
```

```
        public boolean rowUpdate( HashMap row )
        {
                Iterator rowWalker = row.keySet().iterator();
                try
                {
                        Statement sql = main.con.createStatement();
                        while ( rowWalker.hasNext() )
                        {
                               String fieldName = (String)rowWalker.next();
                               String sqlText = "UPDATE " + tableName + " SET " + fieldName + " = '" +
                                                (String)row.get( fieldName ) + "' WHERE " + pkFieldName +
                                                " = '" + (String)rs.getString(pkFieldName) + "'";
                               sql.executeUpdate(sqlText);
                        }

                        Statement stmt = main.con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                                   ResultSet.CONCUR_UPDATABLE );

                        if (tableName == "switch")
                        {
                                rs = stmt.executeQuery( "SELECT * FROM " + tableName + " ORDER BY " +
                                                                                       pkFieldName );
                        }
                        else
                        {
                                rs = stmt.executeQuery( "SELECT * FROM " + tableName + " WHERE " +
                                                fkFieldName + " = '" + fkFieldValue + "' ORDER BY " +
                                                pkFieldName );
                        }

                        rs.absolute( index+1 );
                        fireContentsChanged( this, size-1, size-1 );
                }
                catch ( SQLException e )
                {
                       e.printStackTrace();
                       return false;
                }

               return true;
        }

        public void addPropertyChangeListener( PropertyChangeListener pcl )
        {
                pcs.addPropertyChangeListener( pcl );
        }

        public void addPropertyChangeListener( String propertyName, PropertyChangeListener pcl )
        {
                pcs.addPropertyChangeListener( propertyName, pcl );
        }

        public void removePropertyChangeListener( PropertyChangeListener pcl )
        {
                pcs.removePropertyChangeListener( pcl );
        }

        public void removePropertyChangeListener(String propertyName, PropertyChangeListener pcl )
        {
                pcs.removePropertyChangeListener( propertyName, pcl );
        }
}
```

A.1.5 Klasse PrintUtilities:

```
import java.awt.*;
import javax.swing.*;
import java.awt.print.*;

public class PrintUtilities implements Printable
{
	private Component componentToBePrinted;

	public static void printComponent(Component c)
	{
		new PrintUtilities(c).print();
	}

	public PrintUtilities(Component componentToBePrinted)
	{
		this.componentToBePrinted = componentToBePrinted;
	}

	public void print()
	{
		PrinterJob printJob = PrinterJob.getPrinterJob();
		PageFormat pf = printJob.defaultPage();
		pf.setOrientation(PageFormat.LANDSCAPE);
		printJob.setPrintable(this, pf);
		try
		{
			if(printJob.printDialog())
			{
				printJob.print();
			}
		}
		catch (PrinterException e)
		{
		}
	}
}
```

A.1.6 Klasse PanelSwitch:

```
import java.awt.Dimension;
import javax.swing.JPanel;
import javax.swing.border.Border;
import javax.swing.BorderFactory;
import java.awt.Component;
import javax.swing.*;
import java.sql.SQLException;


public class PanelSwitch extends JPanel
{
        private ImageIcon switchIcon;
        private JLabel switchLabel;
        private JPanel ipswitch;
        private PanelSwitchList psl;
        private PanelSwitchDetails psd;
        private JPanel panelSwitch;
        private ResultSetListModel switchListModel;

        public PanelSwitch( Gui gui )
        {
                super();
                create( gui, null );
        }

        public PanelSwitch( Gui gui, ResultSetListModel rsm )
        {
                super();
                gui.setCurrentPanel( "panelSwitch" );
                create( gui, rsm );
        }

        private void create( Gui gui, ResultSetListModel rsm )
        {
                Border border = BorderFactory.createRaisedBevelBorder();

                switchIcon = new ImageIcon("switches.gif");
                switchLabel = new JLabel(switchIcon);
                ipswitch = new JPanel();
                ipswitch.add(switchLabel);

                ipswitch.setBounds(108,17,79,293);

                if ( rsm != null )
                        psl = new PanelSwitchList( gui, rsm );
                else
                        psl = new PanelSwitchList( gui, new DefaultListModel() );

                psl.setBorder(border);
                psl.setBounds(40,40,220,425);
                psl.setPreferredSize( new Dimension( 200, 425 ) );

                psd = new PanelSwitchDetails( gui, rsm );
                psd.setBorder(border);
                psd.setBounds(260,40,600,425);
                psd.setPreferredSize( new Dimension( 500, 425 ) );

                this.add( ipswitch );
                this.add( psl );
                this.add( psd );
        }

        public PanelSwitchList getPanelSwitchList() { return psl; }
        public PanelSwitchDetails getPanelSwitchDetails() { return psd; }

        public ResultSetListModel getResultSetListModel() { return psl.getResultSetListModel(); }
        public void setResultSetListModel( ResultSetListModel rsm )
        {
                psl.setResultSetListModel( rsm );
                psd.setResultSetListModel( rsm );
        }

}
```

A.1.7 Klasse PanelSwitchList:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Insets;
import javax.swing.*;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.ListModel;
import javax.swing.ListCellRenderer;
import javax.swing.JList;
import javax.swing.ListSelectionModel;
import javax.swing.border.Border;
import javax.swing.BorderFactory;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;

import java.sql.SQLException;


public class PanelSwitchList extends JPanel
{
        private Gui gui;
        private ListModel listModel;
        private ListCellRenderer renderer;
        private JList List;
        private JScrollPane pane;
        public static JPanel panelCabinet;
        private String fkFieldValue;

        public PanelSwitchList( Gui gui, ListModel listModel )
        {
                super( new GridBagLayout() );

                this.listModel = listModel;
                this.gui = gui;

                create();

                addListeners();
        }

        private void create()
        {
                renderer = new NameAndPictureListCellRenderer();

                list = new JList( listModel );
                list.setCellRenderer( renderer );
                list.setVisibleRowCount( 20 );
                list.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );

                pane = new JScrollPane(list);

                GridBagConstraints gbc = new GridBagConstraints(
                        0,0,
                        1,1,
                        0.5,0.5,
                        GridBagConstraints.CENTER,
                        GridBagConstraints.BOTH,
                        new Insets( 0,0,0,0 ),
                        0,0 );

                add( pane, gbc );
        }

        private void addListeners()
        {
                list.addListSelectionListener( new ListSelectionListener()
                {
                        public void valueChanged( ListSelectionEvent e )
                        {
                                ((ResultSetListModel)listModel).setIndex(list.getSelectedIndex() );
                                fkFieldValue = ((ResultSetListModel)listModel).getString("pk_switch");
                        }
                });

                list.addMouseListener( new MouseListener()
                {
                        public void mouseClicked( MouseEvent e ) {
                                if (e.getClickCount() == 2)
                                {
                                        ResultSetListModel cabinetListModel = null;
                                        try
```

```
                                        {
                                                cabinetListModel = new ResultSetListModel( "cabinet",
                                                "pk_cabinet", "cabinet_fkswitch", fkFieldValue );
                                        }
                                        catch ( SQLException exception )
                                        {
                                                exception.printStackTrace();
                                        }

                                        if ( cabinetListModel.getSize() > 0 )
                                        {
                                                gui.removeComponent( "panelSwitch" );
                                                gui.getPanelCabinet().setResultSetListModel(
                                                                                        cabinetListModel );
                                                gui.setCurrentPanel( "panelCabinet" );
                                                gui.getPanelCabinet().setVisible( true );
                                                gui.getContentPane().add(gui.getPanelCabinet(),
                                                                                        BorderLayout.CENTER);
                                                gui.getContentPane().repaint();

                                                Gui.itemUpperLevel.setEnabled(true);
                                                Gui.itemAddSwitch.setEnabled(false);
                                                Gui.itemDelSwitch.setEnabled(false);
                                                Gui.itemAddCabinet.setEnabled(true);
                                                Gui.itemDelCabinet.setEnabled(true);
                                                Gui.itemAddShelf.setEnabled(false);
                                                Gui.itemDelShelf.setEnabled(false);
                                                Gui.itemAddCard.setEnabled(false);
                                                Gui.itemDelCard.setEnabled(false);
                                                Gui.buttonToolBarUpperLevel.setEnabled(true);
                                                Gui.buttonToolBarAddSwitch.setEnabled(false);
                                                Gui.buttonToolBarDelSwitch.setEnabled(false);
                                                Gui.buttonToolBarAddCabinet.setEnabled(true);
                                                Gui.buttonToolBarDelCabinet.setEnabled(true);
                                                Gui.buttonToolBarAddShelf.setEnabled(false);
                                                Gui.buttonToolBarDelShelf.setEnabled(false);
                                                Gui.buttonToolBarAddCard.setEnabled(false);
                                                Gui.buttonToolBarDelCard.setEnabled(false);
                                        }
                                        else
                                        {
                                                System.out.println( "Keine Datensätze vorhanden, Ebene
                                                                                wird nicht gewechselt" );
                                        }
                                }
                        }

                        public void mousePressed(MouseEvent e) {}

                        public void mouseReleased(MouseEvent e) {}

                        public void mouseEntered(MouseEvent e) {}

                        public void mouseExited(MouseEvent e) {}
                });
        }

        public ResultSetListModel getResultSetListModel()
        {
                try
                {
                        return (ResultSetListModel)listModel;
                }
                catch ( ClassCastException e )
                {
                        e.printStackTrace();
                        return null;
                }
        }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                listModel = rsm;
                list.setModel( rsm );
        }
}
```

A.1.8 Klasse PanelSwitchDetails:

```
import java.awt.Insets;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.*;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JButton;
import java.util.HashMap;
import java.util.*;


public class PanelSwitchDetails extends JPanel implements java.beans.PropertyChangeListener
{
        private Gui gui;
        private ResultSetListModel listModel;
        public static JLabel labelPosition;
        public static JLabel labelPositionData;
        private String positionData;
        public static JLabel labelName;
        private JTextField textFieldName;
        public static JLabel labelOwner;
        private JTextField textFieldOwner;
        public static JLabel labelBookingPrime;
        private JTextField textFieldBookingPrime;
        public static JLabel labelProject;
        private JTextField textFieldProject;
        public static JLabel labelOtherProjects;
        private JTextField textFieldOtherProjects;
        public static JLabel labelPurchasingYear;
        private JTextField textFieldPurchasingYear;
        public static JButton buttonSave;
        public static JButton buttonCancel;
        public static JButton buttonDuplicate;
        private HashMap table;

        public PanelSwitchDetails( Gui gui, ResultSetListModel listModel )
        {
                super( new GridBagLayout() );

                this.listModel = listModel;
                this.gui = gui;
                create();
                if ( listModel != null ) {
                        propertyChange( null );
                        listModel.addPropertyChangeListener("index", this);
                }
                addListeners();
        }

        private void create()
        {
                GridBagConstraints gbc = new GridBagConstraints();
                gbc.insets = new Insets(10, 3, 3, 3);
                gbc.anchor = GridBagConstraints.EAST;

                labelPosition = new JLabel(main.getLabelText("currentPosition"));
                gbc.gridx = 0;
                gbc.gridy = 0;
                add(labelPosition, gbc);

                labelPositionData = new JLabel();
                gbc.gridx = 1;
                gbc.gridy = 0;
                add(labelPositionData, gbc);

                labelName = new JLabel(main.getLabelText("name"));
                gbc.gridx = 0;
                gbc.gridy = 1;
                add(labelName, gbc);

                textFieldName = new JTextField(20);
                gbc.gridx = 1;
                gbc.gridy = 1;
                add(textFieldName, gbc);

                labelOwner = new JLabel(main.getLabelText("owner"));
                gbc.gridx = 0;
                gbc.gridy = 2;
```

```
        add(labelOwner, gbc);

        textFieldOwner = new JTextField(20);
        gbc.gridx = 1;
        gbc.gridy = 2;
        add(textFieldOwner, gbc);

        labelBookingPrime = new JLabel(main.getLabelText("bookingPrime"));
        gbc.gridx = 0;
        gbc.gridy = 3;
        add(labelBookingPrime, gbc);

        textFieldBookingPrime = new JTextField(20);
        gbc.gridx = 1;
        gbc.gridy = 3;
        add(textFieldBookingPrime, gbc);

        labelProject = new JLabel(main.getLabelText("project"));
        gbc.gridx = 0;
        gbc.gridy = 4;
        add(labelProject, gbc);

        textFieldProject = new JTextField(20);
        gbc.gridx = 1;
        gbc.gridy = 4;
        add(textFieldProject, gbc);

        labelOtherProjects = new JLabel(main.getLabelText("otherProjects"));
        gbc.gridx = 0;
        gbc.gridy = 5;
        add(labelOtherProjects, gbc);

        textFieldOtherProjects = new JTextField(20);
        gbc.gridx = 1;
        gbc.gridy = 5;
        add(textFieldOtherProjects, gbc);

        labelPurchasingYear = new JLabel(main.getLabelText("purchasingYear"));
        gbc.gridx = 0;
        gbc.gridy = 6;
        add(labelPurchasingYear, gbc);

        textFieldPurchasingYear = new JTextField(20);
        gbc.gridx = 1;
        gbc.gridy = 6;
        add(textFieldPurchasingYear, gbc);

        buttonSave = new JButton(main.getLabelText( "update" ) );
        gbc.gridx = 0;
        gbc.gridy = 7;
        gbc.gridwidth = 2;
        gbc.anchor = GridBagConstraints.WEST;
        gbc.insets = new Insets( 10, 60, 10, 10 );
        add( buttonSave, gbc );

        buttonCancel = new JButton(main.getLabelText( "cancel" ) );
        gbc.gridx = 1;
        gbc.gridy = 7;
        gbc.gridwidth = 2;
        gbc.anchor = GridBagConstraints.EAST;
        gbc.insets = new Insets( 10, 10, 10, 60 );
        add( buttonCancel, gbc );
}

public static void redraw()
{
        labelName.setText(main.getLabelText("name"));
        labelOwner.setText(main.getLabelText("owner"));
        labelBookingPrime.setText(main.getLabelText("bookingPrime"));
        labelProject.setText(main.getLabelText("project"));
        labelOtherProjects.setText(main.getLabelText("otherProjects"));
        labelPurchasingYear.setText(main.getLabelText("purchasingYear"));
        buttonSave.setText(main.getLabelText("update"));
        buttonCancel.setText(main.getLabelText("cancel"));
}

private void addListeners()
{
        buttonSave.addActionListener ( new ActionListener()
        {
                public void actionPerformed ( ActionEvent e )
                {
                        table = new HashMap();
                        table.put("switchname", textFieldName.getText() );
                        table.put("switchowner", textFieldOwner.getText() );
                        table.put("switchbookingprime", textFieldBookingPrime.getText() );
                        table.put("switchproject", textFieldProject.getText() );
```

```
                                table.put("switchotherprojects", textFieldOtherProjects.getText() );
                                table.put("switchpurchasingyear", textFieldPurchasingYear.getText() );

                                ((ResultSetListModel)listModel).rowUpdate(table);
                        }
                });

                buttonCancel.addActionListener ( new ActionListener()
                {
                        public void actionPerformed ( ActionEvent e )
                        {
                                textFieldName.setText(listModel.getString("switchname"));
                                textFieldOwner.setText(listModel.getString("switchowner"));
                                textFieldBookingPrime.setText(listModel.getString("switchbookingprime"));
                                textFieldProject.setText(listModel.getString("switchproject"));

                                textFieldOtherProjects.setText
                                        (listModel.getString("switchotherprojects"));
                                textFieldPurchasingYear.setText
                                        (listModel.getString("switchpurchasingyear"));
                        }
                });

        }

        public void propertyChange( java.beans.PropertyChangeEvent e )
        {
                String text = null;

                main.actualSwitch = ((ResultSetListModel)listModel).getString("switchname");
                positionData = main.actualSwitch;
                labelPositionData.setText( positionData );

                text = listModel.getString("switchname");
                if ( text != null )
                        textFieldName.setText( text );
                else
                        textFieldName.setText( "" );
                text = listModel.getString("switchowner");
                if ( text != null )
                        textFieldOwner.setText( text );
                else
                        textFieldOwner.setText( "" );
                text = listModel.getString("switchbookingprime");
                if ( text != null )
                        textFieldBookingPrime.setText( text );
                else
                        textFieldBookingPrime.setText( "" );
                text = listModel.getString("switchproject");
                if ( text != null )
                        textFieldProject.setText( text );
                else
                        textFieldProject.setText( "" );
                text = listModel.getString("switchotherprojects");
                if ( text != null )
                        textFieldOtherProjects.setText( text );
                else
                        textFieldOtherProjects.setText( "" );
                text = listModel.getString("switchpurchasingyear");
                if ( text != null )
                        textFieldPurchasingYear.setText( text );
                else
                        textFieldPurchasingYear.setText( "" );
        }

        public ResultSetListModel getResultSetListModel() { return listModel; }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                listModel = rsm;
                listModel.addPropertyChangeListener("index", this);
                propertyChange( null );
        }
}
```

A.1.9 Klasse PanelCabinet:

```
import java.awt.Dimension;
import javax.swing.JPanel;
import javax.swing.DefaultListModel;
import javax.swing.border.Border;
import javax.swing.BorderFactory;
import javax.swing.*;
import java.awt.Component;


public class PanelCabinet extends JPanel
{
        private ImageIcon cabinetIcon;
        private JLabel cabinetLabel;
        private JPanel ipcabinet;
        private PanelCabinetList pcabl;
        private PanelCabinetDetails pcabd;

        public PanelCabinet( Gui gui, ResultSetListModel rsm )
        {
                super();

                create( gui, rsm );
        }

        public PanelCabinet( Gui gui )
        {
                super();

                create( gui, null );
        }

        private void create( Gui gui, ResultSetListModel rsm )
        {
                Border border = BorderFactory.createRaisedBevelBorder();

                cabinetIcon = new ImageIcon("cabinets.gif");
                cabinetLabel = new JLabel(cabinetIcon);
                ipcabinet = new JPanel();
                ipcabinet.add(cabinetLabel);

                ipcabinet.setBounds(108,17,79,293);

                if ( rsm != null )
                        pcabl = new PanelCabinetList( gui, rsm );
                else
                        pcabl = new PanelCabinetList(  gui, new DefaultListModel() );
                pcabl.setBorder(border);
                pcabl.setBounds(40,40,220,425);
                pcabl.setPreferredSize( new Dimension( 200, 425 ) );

                pcabd = new PanelCabinetDetails( gui, rsm );
                pcabd.setBorder(border);
                pcabd.setBounds(260,40,600,425);
                pcabd.setPreferredSize( new Dimension( 500, 425 ) );

                this.add( ipcabinet );
                this.add( pcabl );
                this.add( pcabd );
        }

        public PanelCabinetList getPanelCabinetList() { return pcabl; }

        public PanelCabinetDetails getPanelCabinetDetails() { return pcabd; }

        public ResultSetListModel getResultSetListModel() { return pcabl.getResultSetListModel(); }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                pcabl.setResultSetListModel( rsm );
                pcabd.setResultSetListModel( rsm );
        }
}
```

A.1.10 Klasse PanelCabinetList:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Insets;
import javax.swing.*;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.ListModel;
import javax.swing.ListCellRenderer;
import javax.swing.JList;
import javax.swing.ListSelectionModel;
import javax.swing.border.Border;
import javax.swing.BorderFactory;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import java.sql.SQLException;


public class PanelCabinetList extends JPanel
{
        private Gui gui;
        private ListModel listModel;
        private ListCellRenderer renderer;
        private JList list;
        private JScrollPane pane;
        public static JPanel panelShelf;
        private String fkFieldValue;

        public PanelCabinetList( Gui gui, ListModel listModel )
        {
                super( new GridBagLayout() );

                this.listModel = listModel;
                this.gui = gui;

                create();

                addListeners();
        }

        private void create()
        {
                renderer = new NameAndPictureListCellRenderer();

                list = new JList( listModel );
                list.setCellRenderer( renderer );
                list.setVisibleRowCount( 20 );
                list.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );

                pane = new JScrollPane(list);

                GridBagConstraints gbc = new GridBagConstraints(
                        0,0,
                        1,1,
                        0.5,0.5,
                        GridBagConstraints.CENTER,
                        GridBagConstraints.BOTH,
                        new Insets( 0,0,0,0 ),
                        0,0 );

                add( pane, gbc );
        }

        private void addListeners()
        {
                list.addListSelectionListener( new ListSelectionListener()
                {
                        public void valueChanged( ListSelectionEvent e )
                        {
                                ((ResultSetListModel)listModel).setIndex(list.getSelectedIndex() );

                                fkFieldValue = ((ResultSetListModel)listModel).getString("pk_cabinet");
                        }
                });

                list.addMouseListener( new MouseListener()
                {
                        public void mouseClicked( MouseEvent e )
                        {
                                if (e.getClickCount() == 2)
                                {
                                        main.actualCabinet =
                                                ((ResultSetListModel)listModel).getString("cabinetname");
```

```
                                        ResultSetListModel shelfListModel = null;
                                        try
                                        {
                                                shelfListModel = new ResultSetListModel( "shelf",
                                                        "pk_shelf", "shelf_fkcabinet", fkFieldValue);
                                        }
                                        catch ( SQLException exception )
                                        {
                                                exception.printStackTrace();
                                        }

                                        if ( shelfListModel.getSize() > 0 )
                                        {
                                                gui.removeComponent( "panelCabinet" );

                                                gui.getPanelShelf().
                                                        setResultSetListModel( shelfListModel );
                                                gui.setCurrentPanel( "panelShelf" );
                                                gui.getPanelShelf().setVisible(true);
                                                gui.getContentPane().add( gui.getPanelShelf(),
                                                                                BorderLayout.CENTER );
                                                gui.getContentPane().repaint();

                                                Gui.itemAddSwitch.setEnabled(false);
                                                Gui.itemDelSwitch.setEnabled(false);
                                                Gui.itemAddCabinet.setEnabled(false);
                                                Gui.itemDelCabinet.setEnabled(false);
                                                Gui.itemAddShelf.setEnabled(true);
                                                Gui.itemDelShelf.setEnabled(true);
                                                Gui.itemAddCard.setEnabled(false);
                                                Gui.itemDelCard.setEnabled(false);
                                                Gui.buttonToolBarAddSwitch.setEnabled(false);
                                                Gui.buttonToolBarDelSwitch.setEnabled(false);
                                                Gui.buttonToolBarAddCabinet.setEnabled(false);
                                                Gui.buttonToolBarDelCabinet.setEnabled(false);
                                                Gui.buttonToolBarAddShelf.setEnabled(true);
                                                Gui.buttonToolBarDelShelf.setEnabled(true);
                                                Gui.buttonToolBarAddCard.setEnabled(false);
                                                Gui.buttonToolBarDelCard.setEnabled(false);
                                        }
                                        else
                                        {
                                                System.out.println( "Keine Datensaetze vorhanden, Ebene
                                                                        wird nicht gewechselt" );
                                        }
                                }
                        }

                        public void mousePressed(MouseEvent e) {}

                        public void mouseReleased(MouseEvent e) {}

                        public void mouseEntered(MouseEvent e) {}

                        public void mouseExited(MouseEvent e) {}

                });
        }


        public ResultSetListModel getResultSetListModel()
        {
                try
                {
                        return (ResultSetListModel)listModel;
                }
                catch ( ClassCastException e )
                {
                        e.printStackTrace();
                        return null;
                }

        }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                listModel = rsm;
                list.setModel( rsm );
        }
}
```

A.1.11 Klasse PanelCabinetDetails:

```
import java.awt.Insets;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.*;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JButton;
import java.util.HashMap;
import java.util.*;


public class PanelCabinetDetails extends JPanel implements java.beans.PropertyChangeListener
{
        private Gui gui;
        private ResultSetListModel listModel;
        public static JLabel labelPosition;
        public static JLabel labelPositionData;
        private String positionData;
        public static JLabel labelName;
        private JTextField textFieldName;
        public static JLabel labelNumber;
        private JTextField textFieldNumber;
        public static JLabel labelSharingSwitch;
        private JTextField textFieldSharingSwitch;
        public static JLabel labelPlace;
        private JTextField textFieldPlace;
        public static JButton buttonSave;
        public static JButton buttonCancel;
        private HashMap table;

        public PanelCabinetDetails( Gui gui, ResultSetListModel listModel )
        {
                super( new GridBagLayout() );

                this.listModel = listModel;
                this.gui = gui;
                create();
                if ( listModel != null )
                {
                        propertyChange( null );
                        listModel.addPropertyChangeListener("index", this);
                }
                addListeners();

        }

        private void create()
        {
                GridBagConstraints gbc = new GridBagConstraints();
                gbc.insets = new Insets(10, 3, 3, 3);
                gbc.anchor = GridBagConstraints.EAST;

                labelPosition = new JLabel(main.getLabelText("currentPosition"));
                gbc.gridx = 0;
                gbc.gridy = 0;
                add(labelPosition, gbc);

                labelPositionData = new JLabel();
                gbc.gridx = 1;
                gbc.gridy = 0;
                add(labelPositionData , gbc);

                labelName = new JLabel(main.getLabelText("name"));
                gbc.gridx = 0;
                gbc.gridy = 1;
                add(labelName, gbc);

                textFieldName = new JTextField(20);
                gbc.gridx = 1;
                gbc.gridy = 1;
                add(textFieldName, gbc);

                labelNumber = new JLabel(main.getLabelText("number"));
                gbc.gridx = 0;
                gbc.gridy = 2;
                add(labelNumber, gbc);

                textFieldNumber = new JTextField(20);
```

```
		gbc.gridx = 1;
		gbc.gridy = 2;
		add(textFieldNumber, gbc);

		labelSharingSwitch = new JLabel(main.getLabelText("sharingSwitch"));
		gbc.gridx = 0;
		gbc.gridy = 3;
		add(labelSharingSwitch, gbc);

		textFieldSharingSwitch = new JTextField(20);
		gbc.gridx = 1;
		gbc.gridy = 3;
		add(textFieldSharingSwitch, gbc);

		labelPlace = new JLabel(main.getLabelText("place"));
		gbc.gridx = 0;
		gbc.gridy = 4;
		add(labelPlace, gbc);

		textFieldPlace = new JTextField(20);
		gbc.gridx = 1;
		gbc.gridy = 4;
		add(textFieldPlace, gbc);

		buttonSave = new JButton(main.getLabelText( "update" ) );
		gbc.gridx = 0;
		gbc.gridy = 5;
		gbc.gridwidth = 2;
		gbc.anchor = GridBagConstraints.WEST;
		gbc.insets = new Insets( 10, 60, 10, 10 );
		add( buttonSave, gbc );

		buttonCancel = new JButton(main.getLabelText( "cancel" ) );
		gbc.gridx = 1;
		gbc.gridy = 5;
		gbc.gridwidth = 2;
		gbc.anchor = GridBagConstraints.EAST;
		gbc.insets = new Insets( 10, 10, 10, 60 );
		add( buttonCancel, gbc );

}

public static void redraw()
{
		labelName.setText(main.getLabelText("name"));
		labelNumber.setText(main.getLabelText("number"));
		labelSharingSwitch.setText(main.getLabelText("sharingSwitch"));
		labelPlace.setText(main.getLabelText("place"));
		buttonSave.setText(main.getLabelText("update"));
		buttonCancel.setText(main.getLabelText("cancel"));
}

private void addListeners()
{
		buttonSave.addActionListener ( new ActionListener()
		{
			public void actionPerformed ( ActionEvent e )
			{
				table = new HashMap();
				table.put("cabinetname", textFieldName.getText() );
				table.put("cabinetnumber", textFieldNumber.getText() );
				table.put("cabinetsharingswitch", textFieldSharingSwitch.getText() );
				table.put("cabinetplace", textFieldPlace.getText() );

				((ResultSetListModel)listModel).rowUpdate(table);
			}
		});

		buttonCancel.addActionListener ( new ActionListener()
		{
			public void actionPerformed ( ActionEvent e )
			{
				textFieldName.setText(listModel.getString("cabinetname"));
				textFieldNumber.setText(listModel.getString("cabinetnumber"));
				textFieldSharingSwitch.
					setText(listModel.getString("cabinetsharingswitch"));
				textFieldPlace.setText(listModel.getString("cabinetplace"));
			}
		});
}
```

```
        public void propertyChange( java.beans.PropertyChangeEvent e )
        {
                main.actualCabinet = ((ResultSetListModel)listModel).getString("cabinetname");
                positionData = main.actualSwitch + " / " + main.actualCabinet;
                labelPositionData.setText( positionData );

                String text = null;
                text = listModel.getString("cabinetname");
                if ( text != null )
                        textFieldName.setText( text );
                else
                        textFieldName.setText( "" );
                text = listModel.getString("cabinetnumber");
                if ( text != null )
                        textFieldNumber.setText( text );
                else
                        textFieldNumber.setText( "" );
                text = listModel.getString("cabinetsharingswitch");
                if ( text != null )
                        textFieldSharingSwitch.setText( text );
                else
                        textFieldSharingSwitch.setText( "" );
                text = listModel.getString("cabinetplace");
                if ( text != null )
                        textFieldPlace.setText( text );
                else
                        textFieldPlace.setText( "" );
        }

        public ResultSetListModel getResultSetListModel() { return listModel; }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                listModel = rsm;
                listModel.addPropertyChangeListener("index", this);
                propertyChange( null );
        }
}
```

A.1.12 Klasse PanelShelf:

```
import java.awt.Dimension;
import javax.swing.JPanel;
import javax.swing.DefaultListModel;
import javax.swing.border.Border;
import javax.swing.BorderFactory;
import javax.swing.*;
import java.awt.Component;


public class PanelShelf extends JPanel
{
        private ImageIcon shelfIcon;
        private JLabel shelfLabel;
        private JPanel ipshelf;
        private PanelShelfList pshl;
        private PanelShelfDetails pshd;

        public PanelShelf( Gui gui )
        {
                super();

                create( gui, null );
        }

        public PanelShelf( Gui gui, ResultSetListModel rsm )
        {
                super();

                create( gui, rsm );
        }

        private void create( Gui gui, ResultSetListModel rsm )
        {
                Border border = BorderFactory.createRaisedBevelBorder();

                shelfIcon = new ImageIcon("shelves.gif");
                shelfLabel = new JLabel(shelfIcon);
                ipshelf = new JPanel();
                ipshelf.add(shelfLabel);

                ipshelf.setBounds(108,17,79,293);

                if ( rsm != null )
                        pshl = new PanelShelfList( gui, rsm );
                else
                        pshl = new PanelShelfList( gui, new DefaultListModel() );
                pshl.setBorder(border);
                pshl.setBounds(40,40,220,425);
                pshl.setPreferredSize( new Dimension( 200, 425 ) );

                pshd = new PanelShelfDetails( gui, rsm );
                pshd.setBorder(border);
                pshd.setBounds(260,40,600,425);
                pshd.setPreferredSize( new Dimension( 500, 425 ) );

                this.add( ipshelf );
                this.add( pshl );
                this.add( pshd );
        }

        public PanelShelfList getPanelShelfList() { return pshl; }

        public PanelShelfDetails getPanelShelfDetails() { return pshd; }

        public ResultSetListModel getResultSetListModel() { return pshl.getResultSetListModel(); }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                pshl.setResultSetListModel( rsm );
                pshd.setResultSetListModel( rsm );
        }
}
```

A.1.13 Klasse PanelShelfList:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Insets;
import javax.swing.*;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.ListModel;
import javax.swing.ListCellRenderer;
import javax.swing.JList;
import javax.swing.ListSelectionModel;
import javax.swing.border.Border;
import javax.swing.BorderFactory;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import java.sql.SQLException;


public class PanelShelfList extends JPanel
{
        private Gui gui;
        private ListModel listModel;
        private ListCellRenderer renderer;
        private JList List;
        private JScrollPane pane;
        public static JPanel panelCard;
        private String fkFieldValue;

        public PanelShelfList( Gui gui, ListModel listModel )
        {
                super( new GridBagLayout() );

                this.listModel = listModel;
                this.gui = gui;

                create();

                addListeners();
        }

        private void create()
        {
                renderer = new NameAndPictureListCellRenderer();

                list = new JList( listModel );
                list.setCellRenderer( renderer );
                list.setVisibleRowCount( 20 );
                list.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );

                pane = new JScrollPane(list);

                GridBagConstraints gbc = new GridBagConstraints(
                        0,0,
                        1,1,
                        0.5,0.5,
                        GridBagConstraints.CENTER,
                        GridBagConstraints.BOTH,
                        new Insets( 0,0,0,0 ),
                        0,0 );

                add( pane, gbc );
        }

        private void addListeners()
        {
                list.addListSelectionListener( new ListSelectionListener()
                {
                        public void valueChanged( ListSelectionEvent e )
                        {
                                ((ResultSetListModel)listModel).setIndex(list.getSelectedIndex() );
                                fkFieldValue = ((ResultSetListModel)listModel).getString("pk_shelf");
                        }
                });

                list.addMouseListener( new MouseListener()
                {
                        public void mouseClicked( MouseEvent e )
                        {
                                if (e.getClickCount() == 2)
                                {
```

```
                                        main.actualShelf =
                                                ((ResultSetListModel)listModel).getString("shelfname");

                                        ResultSetListModel cardListModel = null;
                                        try
                                        {
                                                cardListModel = new ResultSetListModel( "card",
                                                        "pk_card", "card_fkshelf", fkFieldValue );
                                        }
                                        catch ( SQLException exception )
                                        {
                                                exception.printStackTrace();
                                        }

                                        if ( cardListModel.getSize() > 0 )
                                        {
                                                gui.removeComponent( "panelShelf" );
                                                gui.getPanelCard().
                                                                setResultSetListModel( cardListModel );
                                                gui.setCurrentPanel( "panelCard" );
                                                gui.getPanelCard().setVisible(true);
                                                gui.getContentPane().add(gui.getPanelCard(),
                                                                                BorderLayout.CENTER);
                                                gui.getContentPane().repaint();

                                                Gui.itemAddSwitch.setEnabled(false);
                                                Gui.itemDelSwitch.setEnabled(false);
                                                Gui.itemAddCabinet.setEnabled(false);
                                                Gui.itemDelCabinet.setEnabled(false);
                                                Gui.itemAddShelf.setEnabled(false);
                                                Gui.itemDelShelf.setEnabled(false);
                                                Gui.itemAddCard.setEnabled(true);
                                                Gui.itemDelCard.setEnabled(true);
                                                Gui.buttonToolBarAddSwitch.setEnabled(false);
                                                Gui.buttonToolBarDelSwitch.setEnabled(false);
                                                Gui.buttonToolBarAddCabinet.setEnabled(false);
                                                Gui.buttonToolBarDelCabinet.setEnabled(false);
                                                Gui.buttonToolBarAddShelf.setEnabled(false);
                                                Gui.buttonToolBarDelShelf.setEnabled(false);
                                                Gui.buttonToolBarAddCard.setEnabled(true);
                                                Gui.buttonToolBarDelCard.setEnabled(true);
                                        }
                                }
                        }

                        public void mousePressed(MouseEvent e) {}

                        public void mouseReleased(MouseEvent e) {}

                        public void mouseEntered(MouseEvent e) {}

                        public void mouseExited(MouseEvent e) {}
                });
        }

        public ResultSetListModel getResultSetListModel()
        {
                try
                {
                        return (ResultSetListModel)listModel;
                }
                catch ( ClassCastException e )
                {
                        e.printStackTrace();
                        return null;
                }
        }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                listModel = rsm;
                list.setModel( rsm );
        }
}
```

A.1.14 Klasse PanelShelfDetails:

```
import java.awt.Insets;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.*;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JButton;
import java.util.HashMap;
import java.util.*;


public class PanelShelfDetails extends JPanel implements java.beans.PropertyChangeListener
{
        private Gui gui;
        private ResultSetListModel listModel;
        public static JLabel labelPosition;
        public static JLabel labelPositionData;
        private String positionData;
        public static JLabel labelName;
        private JTextField textFieldName;
        public static JLabel labelNumber;
        private JTextField textFieldNumber;
        public static JButton buttonSave;
        public static JButton buttonCancel;
        private HashMap table;

        public PanelShelfDetails( Gui gui, ResultSetListModel listModel )
        {
                super( new GridBagLayout() );

                this.listModel = listModel;
                this.gui = gui;
                create();
                if ( listModel != null )
                {
                        propertyChange( null );
                        listModel.addPropertyChangeListener("index", this);
                }
                addListeners();
        }

        private void create()
        {
                GridBagConstraints gbc = new GridBagConstraints();
                gbc.insets = new Insets(10, 3, 3, 3);
                gbc.anchor = GridBagConstraints.EAST;

                labelPosition = new JLabel(main.getLabelText("currentPosition"));
                gbc.gridx = 0;
                gbc.gridy = 0;
                add(labelPosition, gbc);

                labelPositionData = new JLabel();
                gbc.gridx = 1;
                gbc.gridy = 0;
                add(labelPositionData , gbc);

                labelName = new JLabel(main.getLabelText("name"));
                gbc.gridx = 0;
                gbc.gridy = 1;
                add(labelName, gbc);

                textFieldName = new JTextField(20);
                gbc.gridx = 1;
                gbc.gridy = 1;
                add(textFieldName, gbc);

                labelNumber = new JLabel(main.getLabelText("number"));
                gbc.gridx = 0;
                gbc.gridy = 2;
                add(labelNumber, gbc);

                textFieldNumber = new JTextField(20);
                gbc.gridx = 1;
                gbc.gridy = 2;
                add(textFieldNumber, gbc);

                buttonSave = new JButton(main.getLabelText( "update" ) );
```

```
                gbc.gridx = 0;
                gbc.gridy = 3;
                gbc.gridwidth = 2;
                gbc.anchor = GridBagConstraints.WEST;
                gbc.insets = new Insets( 10, 60, 10, 10 );
                add( buttonSave, gbc );

                buttonCancel = new JButton(main.getLabelText( "cancel" ) );
                gbc.gridx = 1;
                gbc.gridy = 3;
                gbc.gridwidth = 2;
                gbc.anchor = GridBagConstraints.EAST;
                gbc.insets = new Insets( 10, 10, 10, 60 );
                add( buttonCancel, gbc );
        }

        public static void redraw() {
                labelName.setText(main.getLabelText("name"));
                labelNumber.setText(main.getLabelText("number"));
                buttonSave.setText(main.getLabelText("update"));
                buttonCancel.setText(main.getLabelText("cancel"));
        }

        private void addListeners()
        {
                buttonSave.addActionListener ( new ActionListener()
                {
                        public void actionPerformed ( ActionEvent e )
                        {
                                table = new HashMap();
                                table.put("shelfname", textFieldName.getText() );
                                table.put("shelfnumber", textFieldNumber.getText() );

                                ((ResultSetListModel)listModel).rowUpdate(table);
                        }
                });

                buttonCancel.addActionListener ( new ActionListener()
                {
                        public void actionPerformed ( ActionEvent e )
                        {
                                textFieldName.setText(listModel.getString("shelfname"));
                                textFieldNumber.setText(listModel.getString("shelfnumber"));
                        }
                });
        }

        public void propertyChange( java.beans.PropertyChangeEvent e )
        {
                main.actualShelf = ((ResultSetListModel)listModel).getString("shelfname");
                positionData = main.actualSwitch + " / " + main.actualCabinet + " / " + main.actualShelf;
                labelPositionData.setText( positionData );

                String text = null;
                text = listModel.getString("shelfname");
                if ( text != null )
                        textFieldName.setText( text );
                else
                        textFieldName.setText( "" );
                text = listModel.getString("shelfnumber");
                if ( text != null )
                        textFieldNumber.setText( text );
                else
                        textFieldNumber.setText( "" );
        }

        public ResultSetListModel getResultSetListModel() { return listModel; }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                listModel = rsm;
                listModel.addPropertyChangeListener("index", this);
                propertyChange( null );
        }
}
```

A.1.15 Klasse PanelCard:

```
import java.awt.Dimension;
import javax.swing.DefaultListModel;
import javax.swing.JPanel;
import javax.swing.border.Border;
import javax.swing.BorderFactory;
import javax.swing.*;
import java.awt.Component;


public class PanelCard extends JPanel
{
        private ImageIcon cardIcon;
        private JLabel cardLabel;
        private JPanel ipcard;
        private PanelCardList pcardl;
        private PanelCardDetails pcardd;

        public PanelCard( Gui gui )
        {
                super();

                create( gui, null );
        }

        public PanelCard( Gui gui, ResultSetListModel rsm )
        {
                super();

                create( gui, rsm );
        }

        private void create( Gui gui, ResultSetListModel rsm )
        {
                Border border = BorderFactory.createRaisedBevelBorder();

                cardIcon = new ImageIcon("cards.gif");
                cardLabel = new JLabel(cardIcon);
                ipcard = new JPanel();
                ipcard.add(cardLabel);

                ipcard.setBounds(108,17,79,293);

                if ( rsm != null )
                        pcardl = new PanelCardList( gui, rsm );
                else
                        pcardl = new PanelCardList( gui, new DefaultListModel() );
                pcardl.setBorder(border);
                pcardl.setBounds(40,40,220,425);
                pcardl.setPreferredSize( new Dimension( 200, 425 ) );

                pcardd = new PanelCardDetails( gui, rsm );
                pcardd.setBorder(border);
                pcardd.setBounds(260,40,600,425);
                pcardd.setPreferredSize( new Dimension( 500, 425 ) );

                this.add( ipcard );
                this.add( pcardl );
                this.add( pcardd );
        }

        public PanelCardList getPanelCardList() { return pcardl; }

        public PanelCardDetails getPanelCardDetails() { return pcardd; }

        public ResultSetListModel getResultSetListModel() { return pcardl.getResultSetListModel(); }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                pcardl.setResultSetListModel( rsm );
                pcardd.setResultSetListModel( rsm );
        }
}
```

A.1.16 Klasse PanelCardList:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Insets;
import javax.swing.*;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.ListModel;
import javax.swing.ListCellRenderer;
import javax.swing.JList;
import javax.swing.ListSelectionModel;
import javax.swing.border.Border;
import javax.swing.BorderFactory;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;


public class PanelCardList extends JPanel
{
        private Gui gui;
        private ListModel listModel;
        private ListCellRenderer renderer;
        private JList List;
        private JScrollPane pane;
        private String fkFieldValue;

        public PanelCardList( Gui gui, ListModel listModel )
        {
                super( new GridBagLayout() );

                this.listModel = listModel;
                this.gui = gui;

                create();

                addListeners();
        }

        private void create()
        {
                renderer = new NameAndPictureListCellRenderer();

                list = new JList( listModel );
                list.setCellRenderer( renderer );
                list.setVisibleRowCount( 20 );
                list.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );

                pane = new JScrollPane(list);

                GridBagConstraints gbc = new GridBagConstraints(
                        0,0,
                        1,1,
                        0.5,0.5,
                        GridBagConstraints.CENTER,
                        GridBagConstraints.BOTH,
                        new Insets( 0,0,0,0 ),
                        0,0 );

                add( pane, gbc );
        }

        private void addListeners()
        {
                list.addListSelectionListener( new ListSelectionListener()
                {
                        public void valueChanged( ListSelectionEvent e )
                        {
                                ((ResultSetListModel)listModel).setIndex(list.getSelectedIndex() );
                                fkFieldValue = ((ResultSetListModel)listModel).getString("pk_card");
                        }
                });

                list.addMouseListener( new MouseListener()
                {
                        public void mouseClicked( MouseEvent e )
                        {
                                if (e.getClickCount() == 2)
                                {
                                        main.actualCard =
                                                ((ResultSetListModel)listModel).getString("cardpeccode");
                                }
                        }
```

```
                        public void mousePressed(MouseEvent e) {}

                        public void mouseReleased(MouseEvent e) {}

                        public void mouseEntered(MouseEvent e) {}

                        public void mouseExited(MouseEvent e) {}

                });
        }

        public ResultSetListModel getResultSetListModel()
        {
                try
                {
                        return (ResultSetListModel)listModel;
                }
                catch ( ClassCastException e )
                {
                        e.printStackTrace();
                        return null;
                }

        }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                listModel = rsm;
                list.setModel( rsm );
        }
}
```

A.1.17 Klasse PanelCardDetails:

```
import java.awt.Insets;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.*;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JButton;
import java.util.HashMap;
import java.util.*;


public class PanelCardDetails extends JPanel implements java.beans.PropertyChangeListener
{
        private Gui gui;
        private ResultSetListModel listModel;
        public static JLabel labelPosition;
        public static JLabel labelPositionData;
        private String positionData;
        public static JLabel labelPECCode;
        private JTextField textFieldPECCode;
        public static JLabel labelMDDate;
        private JTextField textFieldMDDate;
        public static JLabel labelReplacement;
        private JTextField textFieldReplacement;
        public static JLabel labelHardwareRelease;
        private JTextField textFieldHardwareRelease;
        public static JLabel labelLiability;
        private JTextField textFieldLiability;
        public static JLabel labelConnection;
        private JTextField textFieldConnection;
        public static JLabel labelSpareNumber;
        private JTextField textFieldSpareNumber;
        public static JLabel labelSoftwareRelease;
        private JTextField textFieldSoftwareRelease;
        public static JLabel labelPanelSide;
        private JTextField textFieldPanelSide;

        public static JButton buttonSave;
        public static JButton buttonCancel;
        private HashMap table;

        public PanelCardDetails( Gui gui, ResultSetListModel listModel )
        {
                super( new GridBagLayout() );

                this.listModel = listModel;
                this.gui = gui;
                create();
                if ( listModel != null )
                {
                        propertyChange( null );
                        listModel.addPropertyChangeListener("index", this);
                }
                addListeners();

        }

        private void create()
        {
                GridBagConstraints gbc = new GridBagConstraints();
                gbc.insets = new Insets(10, 3, 3, 3);
                gbc.anchor = GridBagConstraints.EAST;

                labelPosition = new JLabel(main.getLabelText("currentPosition"));
                gbc.gridx = 0;
                gbc.gridy = 0;
                add(labelPosition, gbc);

                labelPositionData = new JLabel();
                gbc.gridx = 1;
                gbc.gridy = 0;
                add(labelPositionData , gbc);

                labelPECCode = new JLabel(main.getLabelText("PECCode"));
                gbc.gridx = 0;
                gbc.gridy = 1;
                add(labelPECCode, gbc);
```

```
textFieldPECCode = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 1;
add(textFieldPECCode, gbc);

labelMDDate = new JLabel(main.getLabelText("MDDate"));
gbc.gridx = 0;
gbc.gridy = 2;
add(labelMDDate, gbc);

textFieldMDDate = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 2;
add(textFieldMDDate, gbc);

labelReplacement = new JLabel(main.getLabelText("replacement"));
gbc.gridx = 0;
gbc.gridy = 3;
add(labelReplacement, gbc);

textFieldReplacement = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 3;
add(textFieldReplacement, gbc);

labelHardwareRelease = new JLabel(main.getLabelText("hardwareRelease"));
gbc.gridx = 0;
gbc.gridy = 4;
add(labelHardwareRelease, gbc);

textFieldHardwareRelease = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 4;
add(textFieldHardwareRelease, gbc);

labelLiability = new JLabel(main.getLabelText("liability"));
gbc.gridx = 0;
gbc.gridy = 5;
add(labelLiability, gbc);

textFieldLiability = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 5;
add(textFieldLiability, gbc);

labelConnection = new JLabel(main.getLabelText("connection"));
gbc.gridx = 0;
gbc.gridy = 6;
add(labelConnection, gbc);

textFieldConnection = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 6;
add(textFieldConnection, gbc);

labelSpareNumber = new JLabel(main.getLabelText("spareNumber"));
gbc.gridx = 0;
gbc.gridy = 7;
add(labelSpareNumber, gbc);

textFieldSpareNumber = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 7;
add(textFieldSpareNumber, gbc);

labelSoftwareRelease = new JLabel(main.getLabelText("softwareRelease"));
gbc.gridx = 0;
gbc.gridy = 8;
add(labelSoftwareRelease, gbc);

textFieldSoftwareRelease = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 8;
add(textFieldSoftwareRelease, gbc);

labelPanelSide = new JLabel(main.getLabelText( "panelside" ) );
gbc.gridx = 0;
gbc.gridy = 9;
add(labelPanelSide, gbc);

textFieldPanelSide = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 9;
add(textFieldPanelSide, gbc);

buttonSave = new JButton(main.getLabelText( "update" ) );
```

```
        gbc.gridx = 0;
        gbc.gridy = 10;
        gbc.gridwidth = 2;
        gbc.anchor = GridBagConstraints.WEST;
        gbc.insets = new Insets( 10, 60, 10, 10 );
        add( buttonSave, gbc );

        buttonCancel = new JButton(main.getLabelText( "cancel" ) );
        gbc.gridx = 1;
        gbc.gridy = 10;
        gbc.gridwidth = 2;
        gbc.anchor = GridBagConstraints.EAST;
        gbc.insets = new Insets( 10, 10, 10, 60 );
        add( buttonCancel, gbc );
}

public static void redraw()
{
        labelPECCode.setText(main.getLabelText("PECCode"));
        labelMDDate.setText(main.getLabelText("MDDate"));
        labelReplacement.setText(main.getLabelText("replacement"));
        labelHardwareRelease.setText(main.getLabelText("hardwareRelease"));
        labelLiability.setText(main.getLabelText("liability"));
        labelConnection.setText(main.getLabelText("connection"));
        labelSpareNumber.setText(main.getLabelText("spareNumber"));
        labelSoftwareRelease.setText(main.getLabelText("softwareRelease"));
        labelPanelSide.setText(main.getLabelText("panelSide"));
        buttonSave.setText(main.getLabelText("update"));
        buttonCancel.setText(main.getLabelText("cancel"));
}

private void addListeners()
{
        buttonSave.addActionListener ( new ActionListener()
        {
                public void actionPerformed ( ActionEvent e )
                {
                        table = new HashMap();
                        table.put("cardpeccode", textFieldPECCode.getText() );
                        table.put("cardmddate", textFieldMDDate.getText() );
                        table.put("cardreplacement", textFieldReplacement.getText() );
                        table.put("cardhardwarerelease", textFieldHardwareRelease.getText() );
                        table.put("cardliability", textFieldLiability.getText() );
                        table.put("cardconnection", textFieldConnection.getText() );
                        table.put("cardsparenumber", textFieldSpareNumber.getText() );
                        table.put("cardsoftwarerelease", textFieldSoftwareRelease.getText() );
                        table.put("cardpanelside", textFieldPanelSide.getText() );

                        ((ResultSetListModel)listModel).rowUpdate(table);
                }
        });

        buttonCancel.addActionListener ( new ActionListener()
        {
                public void actionPerformed ( ActionEvent e )
                {
                        textFieldPECCode.setText(listModel.getString("cardpeccode"));
                        textFieldMDDate.setText(listModel.getString("cardmddate"));
                        textFieldReplacement.setText(listModel.getString("cardreplacement"));
                        textFieldHardwareRelease.
                                        setText(listModel.getString("cardhardwarerelease"));
                        textFieldLiability.setText(listModel.getString("cardliability"));
                        textFieldConnection.setText(listModel.getString("cardconnection"));
                        textFieldSpareNumber.setText(listModel.getString("cardsparenumber"));
                        textFieldSoftwareRelease.
                                        setText(listModel.getString("cardsoftwarerelease"));
                        textFieldPanelSide.setText(listModel.getString("cardpanelside"));
                }
        });
}


public void propertyChange( java.beans.PropertyChangeEvent e )
{
        main.actualCard = ((ResultSetListModel)listModel).getString("cardpeccode");
        positionData = main.actualSwitch + " / " + main.actualCabinet + " / " + main.actualShelf +
                                                                " / " + main.actualCard;
        labelPositionData.setText( positionData );

        String text = null;
        text = listModel.getString("cardpeccode");
        if ( text != null )
                textFieldPECCode.setText( text );
        else
                textFieldPECCode.setText( "" );
        text = listModel.getString("cardmddate");
        if ( text != null )
```

```
                        textFieldMDDate.setText( text );
                else
                        textFieldMDDate.setText( "" );
                text = listModel.getString("cardreplacement");
                if ( text != null )
                        textFieldReplacement.setText( text );
                else
                        textFieldReplacement.setText( "" );
                text = listModel.getString("cardhardwarerelease");
                if ( text != null )
                        textFieldHardwareRelease.setText( text );
                else
                        textFieldHardwareRelease.setText( "" );
                text = listModel.getString("cardliability");
                if ( text != null )
                        textFieldLiability.setText( text );
                else
                        textFieldLiability.setText( "" );
                text = listModel.getString("cardconnection");
                if ( text != null )
                        textFieldConnection.setText( text );
                else
                        textFieldConnection.setText( "" );
                text = listModel.getString("cardsparenumber");
                if ( text != null )
                        textFieldSpareNumber.setText( text );
                else
                        textFieldSpareNumber.setText( "" );
                text = listModel.getString("cardsoftwarerelease");
                if ( text != null )
                        textFieldSoftwareRelease.setText( text );
                else
                        textFieldSoftwareRelease.setText( "" );
                text = listModel.getString("cardpanelside");
                if ( text != null )
                        textFieldPanelSide.setText( text );
                else
                        textFieldPanelSide.setText( "" );
        }

        public ResultSetListModel getResultSetListModel() { return listModel; }

        public void setResultSetListModel( ResultSetListModel rsm )
        {
                listModel = rsm;
                listModel.addPropertyChangeListener("index", this);
                propertyChange( null );
        }
}
```

A.2 Klassenbeschreibungen

Für die Realisierung der Applikation werden die folgenden Klassen verwendet.

A.2.1 Klasse Main

Beschreibung:

Diese Klasse stellt die Hauptklasse der Anwendung dar und dient der Herstellung einer Verbindung zum Datenbankmanagementsystem sowie der Festlegung der Sprache und des *Look & Feels*, mit denen die Anwendung startet. Anschliessend wird das Hauptfenster erzeugt und mittels der Erzeugung eines Objektes vom Typ *Gui* die Rahmenstruktur der Anwendung definiert und diese in ihren Ausgangszustand gebracht.

Attribute:

Connection con	Objekt, das für die Verbindung zur Datenbank zuständig ist.
Gui objGui	Objekt, das die Rahmenstruktur festlegt.
String database	String, der den Namen der Datenbank enthält.
String username	String, der den Benutzernamen enthält.
String password	String, der das Passwort enthält.
String language	String, der die Sprache enthält.
String country	String, der das Land enthält.
Locale currentLocale	Objekt, das Lokalisierungsdaten beinhaltet.
ResourceBundle Messages	Objekt, das die in Textdateien ausgelagerten Zeichenketten für die Beschriftung von Benutzeroberflächen-Elementen bündelt.
JFrame frame	Rahmen der Anwendung.

Methoden:

void main()	Methode für die Festlegung der Sprache und des *Look & Feels* sowie die Erzeugung der Rahmenstruktur der Anwendung.

A.2.2 Klasse Gui

Beschreibung:

Diese Klasse dient der Erstellung der Rahmenstruktur der Benutzeroberfläche mit ihrer *MenuBar*, *ToolBar* und ihren Panels. Des Weiteren werden in dieser Klasse allen Schaltelementen *ActionListener* zugewiesen, um auf Ereignisse reagieren und die jeweiligen Methoden aufrufen zu können.

Attribute:

PanelSwitch panelSwitch	Panel, das alle für Switches relevanten Daten enthält.
PanelCabinet panelCabinet	Panel, das alle für Cabinets relevanten Daten enthält.
PanelShelf panelShelf	Panel, das alle für Shelves relevanten Daten enthält.
PanelCard panelCard	Panel, das alle für Cards relevanten Daten enthält.
Icon iconUpperLevel	Bild, das die Funktion zur Navigation in eine höhere Ebene repräsentiert.
Icon iconAddSwitch	Bild, das die Funktion zur Hinzufügung eines Switches in die Datenbank repräsentiert.
Icon iconDelSwitch	Bild, das die Funktion zur Entfernung eines Switches aus der Datenbank repräsentiert.
Icon iconAddCabinet	Bild, das die Funktion zur Hinzufügung eines Cabinets in die Datenbank repräsentiert.
Icon iconDelCabinet	Bild, das die Funktion zur Entfernung eines Cabinets aus der Datenbank repräsentiert.
Icon iconAddShelf	Bild, das die Funktion zur Hinzufügung eines Shelfs in die Datenbank repräsentiert.
Icon iconDelShelf	Bild, das die Funktion zur Entfernung eines Shelfs aus der Datenbank repräsentiert.
Icon iconAddCard	Bild, das die Funktion zur Hinzufügung einer Card in die Datenbank repräsentiert.

Icon iconDelCard	Bild, das die Funktion zur Entfernung einer Card aus der Datenbank repräsentiert.
Icon iconPrintConfig	Bild, das die Funktion des Druckens repräsentiert.
Icon iconInfo	Bild, das die Funktion der Ausgabe von Programminformation repräsentiert.
Icon iconExit	Bild, das die Funktion zum Verlassen der Anwendung repräsentiert.
JMenuBar menuBar	Menüleiste, die alle Funktionen enthält.
JMenu functionsMenu	Menü innerhalb der Menüleiste mit den grundlegenden Funktionen.
JMenu lookAndFeelMenu	Menü innerhalb der Menüleiste für die Festlegung des *Look & Feels.*
JMenu languageMenu	Menü innerhalb der Menüleiste für die Festlegung der Sprache.
JMenu helpMenu	Menü innerhalb der Menüleiste für den Aufruf von Programminformation.
JMenuItem itemUpperLevel	Menüeintrag innerhalb des Menüs *Functions* für die Navigation in eine höhere Ebene.
JMenuItem itemAddSwitch	Menüeintrag innerhalb des Menüs *Functions* für das Hinzufügen eines Switches.
JMenuItem itemDelSwitch	Menüeintrag innerhalb des Menüs *Functions* für das Entfernen eines Switches.
JMenuItem itemAddCabinet	Menüeintrag innerhalb des Menüs *Functions* für das Hinzufügen eines Cabinets.
JMenuItem itemDelCabinet	Menüeintrag innerhalb des Menüs *Functions* für das Entfernen eines Cabinets.
JMenuItem itemAddShelf	Menüeintrag innerhalb des Menüs *Functions* für das Hinzufügen eines Shelfs.
JMenuItem itemDelShelf	Menüeintrag innerhalb des Menüs *Functions* für das Entfernen eines Shelfs.
JMenuItem itemAddCard	Menüeintrag innerhalb des Menüs *Functions* für das Hinzufügen einer Card.

JMenuItem itemDelCard	Menüeintrag innerhalb des Menüs *Functions* für das Entfernen einer Card.
JMenuItem itemPrintConfig	Menüeintrag innerhalb des Menüs *Functions* für das Drucken.
JMenuItem itemExit	Menüeintrag innerhalb des Menüs *Functions* für das Verlassen der Anwendung.
ButtonGroup groupLookAndFeel	Gruppe von *RadioButtons*, die für die Festlegung des *Look & Feels* zuständig sind .
JRadioButtonMenuItem itemJavaLookAndFeel	Menüeintrag innerhalb des Menüs *Look & Fee*l für die Festlegung des *JAVA-Look & Feels.*
JRadioButtonMenuItem itemMotifLookAndFeel	Menüeintrag innerhalb des Menüs *Look & Feel* für die Festlegung des *Motif-Look & Feels.*
JRadioButtonMenuItem itemWindowsStyleLookAndFeel	Menüeintrag innerhalb des Menüs *Look & Feel* für die Festlegung des *Windows-Look & Feels.*
ButtonGroup groupLanguage	Gruppe von *RadioButtons*, die für die Festlegung der Sprache zuständig sind.
JRadioButtonMenuItem itemEnglish	Menüeintrag innerhalb des Menüs *Language* für die Festlegung der englischen Sprache.
JRadioButtonMenuItem itemGerman	Menüeintrag innerhalb des Menüs *Language* für die Festlegung der deutschen Sprache.
JRadioButtonMenuItem itemFrench	Menüeintrag innerhalb des Menüs *Language* für die Festlegung der französischen Sprache.
JMenuItem itemInfo	Menüeintrag innerhalb des Menüs *Help* für den Aufruf von Programminformation.
JToolBar toolBar	*ToolBar*, die den schnellen Zugriff auf die wichtigsten Funktionen ermöglicht.
JButton buttonToolBarUpperLevel	Schaltfläche innerhalb der *ToolBar*, die für die Navigation in eine höhere Ebene zuständig ist.
JButton buttonToolBarAddSwitch	Schaltfläche innerhalb der *ToolBar*, die für das Hinzufügen eines Switches zuständig ist.
JButton buttonToolBarDelSwitch	Schaltfläche innerhalb der *ToolBar*, die für das Entfernen eines Switches zuständig ist.

JButton buttonToolBarAddCabinet	Schaltfläche innerhalb der *ToolBar*, die für das Hinzufügen eines Cabinets zuständig ist.
JButton buttonToolBarDelCabinet	Schaltfläche innerhalb der *ToolBar*, die für das Entfernen eines Cabinets zuständig ist.
JButton buttonToolBarAddShelf	Schaltfläche innerhalb der *ToolBar*, die für das Hinzufügen eines Shelfs zuständig ist.
JButton buttonToolBarDelShelf	Schaltfläche innerhalb der *ToolBar*, die für das Entfernen eines Shelfs zuständig ist.
JButton buttonToolBarAddCard	Schaltfläche innerhalb der *ToolBar*, die für das Hinzufügen einer Card zuständig ist.
JButton buttonToolBarDelCard	Schaltfläche innerhalb der *ToolBar*, die für das Entfernen einer Card zuständig ist.
JButton buttonToolBarPrintConfig	Schaltfläche innerhalb der *ToolBar*, die für das Drucken zuständig ist.
JButton buttonToolBarInfo	Schaltfläche innerhalb der *ToolBar*, die für die Ausgabe von Programminformation zuständig ist.
JButton buttonToolBarExit	Schaltfläche innerhalb der *ToolBar*, die für das Verlassen der Anwendung zuständig ist.
ResultSetListModel switchListModel	Objekt, das die Ergebnisdatensätze beim Auslesen von Information bezüglich Switches aus der Datenbank beinhaltet.
Container contentPane	Objekt, das alle Elemente der Benutzeroberfläche beinhaltet.

Methoden:

Gui()	Konstruktor, der den String in der Titelleiste setzt, die Methode *createGUI()* für die Generierung der Benutzeroberfläche und die Methode *addListeners()* für die Zuweisung der *ActionListener* zu den einzelnen Benutzeroberflächen-Elementen aufruft.
void createGUI()	Methode, die die Benutzeroberfläche mit all ihren Elementen generiert.
void addListeners()	Methode, die die *ActionListener* den einzelnen Benutzeroberflächen-Elementen zuordnet.
void upperLevelClick()	Methode, die die Navigation in eine höhere Ebene ermöglicht und entsprechend die Panels in ihrer Sichtbarkeit sowie die Schaltflächen in ihrer Möglichkeit der Aktivierung setzt.
void addSwitchClick()	Methode, die ein Dialogfenster öffnet und die Funktion *addElement()* der Klasse *ResultSetListModel* für das Hinzufügen eines Switches in die Datenbank aufruft.
void delSwitchClick()	Methode, die ein Dialogfenster öffnet und die Funktion *remove()* der Klasse *ResultSetListModel* für das Entfernen eines Switches aus der Datenbank aufruft.
void addCabinetClick()	Methode, die ein Dialogfenster öffnet und die Funktion *addElement()* der Klasse *ResultSetListModel* für das Hinzufügen eines Cabinets in die Datenbank aufruft.
void delCabinetClick()	Methode, die ein Dialogfenster öffnet und die Funktion *remove()* der Klasse *ResultSetListModel* für das Entfernen eines Cabinets aus der Datenbank aufruft.

void addShelfClick()	Methode, die ein Dialogfenster öffnet und die Funktion *addElement()* der Klasse *ResultSetListModel* für das Hinzufügen eines Shelfs in die Datenbank aufruft.
void delShelfClick()	Methode, die ein Dialogfenster öffnet und die Funktion *remove()* der Klasse *ResultSetListModel* für das Entfernen eines Shelfs aus der Datenbank aufruft.
void addCardClick()	Methode, die ein Dialogfenster öffnet und die Funktion *addElement()* der Klasse *ResultSetListModel* für das Hinzufügen einer Card in die Datenbank aufruft.
void delCardClick()	Methode, die ein Dialogfenster öffnet und die Funktion *remove()* der Klasse *ResultSetListModel* für das Entfernen einer Card aus der Datenbank aufruft.
String addFirstCabinet()	Methode, die ein Dialogfenster öffnet und für das Hinzufügen eines Cabinets in die Datenbank aufgerufen wird, wenn noch keine Cabinets für diesen Switch existieren.
String addFirstShelf()	Methode, die ein Dialogfenster öffnet und für das Hinzufügen eines Shelfs in die Datenbank aufgerufen wird, wenn noch keine Shelves für dieses Cabinet existieren.
String addFirstCard()	Methode, die ein Dialogfenster öffnet und für das Hinzufügen einer Card in die Datenbank aufgerufen wird, wenn noch keine Cards für dieses Shelf existieren.
void printConfigClick()	Methode, die durch Aufruf der Funktion *printComponent()* der Klasse *PrintUtilities* das Drucker-Dialogfenster öffnet und das Drucken der Konfigurationsdaten veranlässt.

void infoClick()	Methode, die ein Dialogfenster öffnet und Programminformationen ausgibt.
void exitClick()	Methode, die ein Dialogfenster öffnet und das Verlassen der Anwendung ermöglicht.
void lookAndFeelClick()	Methode, die es möglich macht, das *Look & Feel* der Anwendung zu wechseln.
void languageClick()	Methode, die es möglich macht, die Sprache der Anwendung zu wechseln.
String getLabelText()	Methode für die Lieferung des jeweiligen Strings für die Beschriftung der Benutzeroberflächen-Elemente, abhängig von den aktuell eingestellten Lokalisierungsdaten.
void setCurrentPanel()	Methode zur Festlegung des aktuellen Panels.
String getCurrentPanel()	Methode zur Abfrage des aktuellen Panels.
PanelSwitch getPanelSwitch()	Methode zur Lieferung des Panels *PanelSwitch.*
PanelCabinet getPanelCabinet()	Methode zur Lieferung des Panels *PanelCabinet.*
PanelShelf getPanelShelf()	Methode zur Lieferung des Panels *PanelShelf.*
PanelCard getPanelCard()	Methode zur Lieferung des Panels *PanelCard.*
void removeComponent()	Methode zur Entfernung von Elementen der Benutzeroberfläche.

A.2.3 Klasse NameAndPictureListCellRenderer

Beschreibung:

Diese Klasse dient lediglich der Definition der Umrahmung eines selektierten Listenelements.

Attribute:

Border lineBorder	Objekt, das die Umrahmung eines fokussierten Listenelements definiert.
Border emptyBorder	Objekt, das die Umrahmung eines nicht fokussierten Listenelements definiert.

Methoden:

NameAndPictureListCell Renderer()	Konstruktor.
Component getListCellRendererComponent()	Methode, die Listenelemente auf ihre Selektion überprüft und diese entsprechend darstellt.

A.2.4 Klasse ResultSetListModel

Beschreibung:

Mittels dieser Klasse wird der Zugriff auf die Datenbank realisiert und über diverse Methoden eine Art Schnittstelle zur Verfügung gestellt, die es erlaubt, Datensätze zu lesen, der Datenbank Elemente hinzuzufügen, zu modifizieren und zu entfernen.

Attribute:

int index	Variable, die die jeweilige Position innerhalb des Ergebnisdatensatzes (*ResultSet*) speichert.
int size	Variable, die die Grösse des Ergebnisdatensatzes (*ResultSet*) speichert.
String tableName	String, der den Namen der Relation enthält.
String pkFieldName	String, der den Namen des Primärschlüssels enthält.
String fkFieldName	String, der den Namen des Fremdschlüssels enthält.
String fkFieldValue	String, der den Wert des Fremdschlüssels enthält.
PropertyChangeSupport pcs	Objekt, das das Management der *PropertyChangeListener* übernimmt.
ResultSet rs	Objekt, das einen Ergebnisdatensatz speichert.
Statement sql	Objekt, das ein *SQL-Statement* speichert.

Methoden:

ResultSetListModel()	Konstruktor, der die jeweilige Relation aus der Datenbank liest, diese als Ergebnisdatensatz speichert, die Grösse bestimmt und den Index initialisiert.
boolean setIndex()	Methode, die den Index innerhalb des Ergebnisdatensatzes neu setzt.
int getIndex()	Methode, die den aktuellen Index des Ergebnisdatensatzes zurückliefert.
void setSize()	Methode, die die Grösse des Ergebnisdatensatzes neu setzt.
int getSize()	Methode, die die Grösse des Ergebnisdatensatzes liefert.
Object getElementAt()	Methode, die ein Objekt aus dem Ergebnisdatensatz in Abhängigkeit des übergebenen Index zurückgibt.
String getElementAtForList()	Methode, die den Facility-Namen als String aus dem Ergebnisdatensatz in Abhängigkeit des übergebenen Index zurückgibt.
String getString()	Methode, die einen String aus dem Ergebnisdatensatz in Abhängigkeit des übergebenen Spaltennamens zurückgibt.
boolean addElement()	Methode, die der jeweiligen Relation ein übergebenes Objekt hinzufügt.
boolean remove()	Methode, die in der jeweiligen Relation abhängig vom übergebenen Index ein Element löscht.
boolean rowUpdate()	Methode, die alle in Form einer *HashMap* (Tabelle) übergebenen Daten in der Relation aktualisiert.
void addPropertyChangeListener()	Methode, die der Registrierung von *Listenern* am *ResultSetListModel* dient.

void removePropertyChangeListener()	Methode, die der Deregistrierung von *Listenern* am *ResultSetListModel* dient.

A.2.5 Klasse PrintUtilities

Beschreibung:

Diese Klasse ist zuständig für das Drucken von Facilities und ihren zugehörigen Attributen.

Attribute:

Component componentToBePrinted	Komponente, die zum Drucken übergeben wird.

Methoden:

PrintUtilities()	Konstruktor.
printComponent()	Methode, die ein Objekt der Klasse erzeugt und auf dieses die Methode *print()* anwendet.
print()	Methode, die das Format des Drucks festlegt, ein Druckdialogfenster öffnet und einen Druckauftrag abschickt.

A.2.6 Klasse PanelSwitch

Beschreibung:

Mit der Erzeugung eines Objektes dieser Klasse werden aus dem Panel *PanelSwitch* heraus drei Sub-Panels mit den Namen *PanelSwitchImage*, *PanelSwitchList* und *PanelSwitchDetails* erzeugt. Diese wiederum repräsentieren alle für Switches relevanten Daten.

Attribute:

ImageIcon switchIcon	Bild, das die auf der jeweiligen Ebene zu verwaltenden Facilities repräsentiert.
JLabel switchLabel	*Label*, welches das *switchIcon* einbettet.
JPanel PanelSwitchImage	Sub-Panel, welches das *switchLabel* und *switchIcon* einbettet.
PanelSwitchList psl	Sub-Panel, welches die Liste mit Facilities vom Typ Switch enthält.
PanelSwitchDetails psd	Sub-Panel, welches die Attribute der in der Liste selektierten Facilities anzeigt.
JPanel panelSwitch	Panel, welches die Sub-Panels mit den Namen *PanelSwitchImage*, *PanelSwitchList* und *PanelSwitchDetails* enthält.
ResultSetListModel switchListModel	Objekt, das die Ergebnisdatensätze beim Auslesen von Information bezüglich Switches aus der Datenbank beinhaltet.

Methoden:

PanelSwitch()	Konstruktor, der den Panel *PanelSwitch* als aktuellen Panel festlegt und die Methode *create()* für die Erzeugung der Sub-Panels aufruft.
void create()	Methode, die für die Erstellung der Sub-Panels verantwortlich ist.
PanelSwitchList getPanelSwitchList()	Methode, welche ein Objekt vom Typ *PanelSwitchList* zurückliefert.
PanelSwitchDetails getPanelSwitchDetails()	Methode, welche ein Objekt vom Typ *PanelSwitchDetails* zurückliefert.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Switches zurückliefert.
ResultSetListModel setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Switches gültigen Ergebnisdatensatz erklärt.

A.2.7 Klasse PanelSwitchList

Beschreibung:

Diese Klasse ist zuständig für die listenförmige Darstellung der im Ergebnisdatensatz vorliegenden Namen von Switches innerhalb eines eigenen Panels sowie für die Behandlung von Ereignissen mit der Maus.

Attribute:

Gui gui	Objekt, das die Rahmenstruktur festlegt.
ListModel listModel	Objekt, das Datensätze in Form einer Liste enthält.
ListCellRenderer renderer	Objekt, das ein selektiertes Listenelement repräsentiert.
JList list	Objekt, das eine Liste repräsentiert.
JScrollPane pane	Panel, das eine *ScrollBar* beinhaltet.
JPanel panelCabinet	Panel, welches die Sub-Panels mit den Namen *PanelCabinetImage*, *PanelCabinetList* und *PanelCabinetDetails* enthält.
String fkFieldValue	String, der den Wert des Fremdschlüssels enthält.

Methoden:

PanelSwitchList()	Konstruktor, der mittels dem übergebenen *ListModel* eine Liste mit Datensätzen von Switches erzeugt.
void create()	Methode zur Erstellung einer Liste mit einer *ScrollBar.*
void addListeners()	Methode, die für die Zuweisung der *ActionListener* zu den einzelnen Listenelementen zuständig ist.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Switches zurückliefert.

void setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Switches gültigen Ergebnisdatensatz erklärt.

A.2.8 Klasse PanelSwitchDetails

Beschreibung:

Diese Klasse repräsentiert die Attribute der selektierten Switches in der Liste. Sie fungiert als *Observer* und reagiert mit einer Aktualisierung der Attributwerte, wenn eine Änderung bezüglich des *Subjects* aufgetreten ist.

Attribute:

Gui gui	Objekt, das die Rahmenstruktur festlegt.
ResultSetListModel listModel	Objekt, das die Ergebnisdatensätze beim Auslesen von Information bezüglich Switches aus der Datenbank beinhaltet.
JLabel labelPosition	*Label* Current Position.
JLabel labelPositionData	*Label*, das die aktuelle Position innerhalb der baumförmigen Struktur der Datenbank anzeigt.
String positionData	Daten des *Labels*, das die aktuelle Position innerhalb der baumförmigen Struktur der Datenbank anzeigt.
JLabel labelName	*Label* Name.
JTextField textFieldName	*TextField*, das Daten bezüglich dem Namen des Switches enthält.
JLabel labelOwner	*Label* Owner.
JTextField textFieldOwner	*TextField*, das Daten bezüglich dem Besitzer des Switches enthält.
JLabel labelBookingPrime	*Label* Booking Prime.
JTextField textFieldBookingPrime	*TextField*, das Daten bezüglich dem Verantwortlichen des Switches enthält.
JLabel labelProject	*Label* Project

JTextField textFieldProject	*TextField*, das Daten bezüglich dem Projekt des Switches enthält.
JLabel labelOtherProjects	*Label* Other Projects.
JTextField textFieldOtherProjects	*TextField*, das Daten bezüglich anderer Projekte des Switches enthält.
JLabel labelPurchasingYear	*Label* Purchasing Year.
JTextField textFieldPurchasingYear	*TextField*, das Daten bezüglich dem Anschaffungsdatum des Switches enthält.
JButton buttonSave	Schaltfläche, welche die Übernahme der in den *TextFields* befindlichen Daten in die Datenbank auslöst.
JButton buttonCancel	Schaltfläche, welche Modifikationen in den *TextFields* rückgängig macht.
HashMap table	Tabelle, welche für die Übernahme der in den *TextFields* befindlichen Daten in die Datenbank benötigt wird.

Methoden:

PanelSwitchDetails()	Konstruktor, der mittels der Methode *create()* für die Erzeugung der *TextFields* und Schaltflächen zuständig ist sowie diesen die *ActionListener* zuweist, um auf Ereignisse reagieren zu können.
void create()	Methode, welche die *TextFields* für die Attributwerte sowie Schaltflächen für das Auslösen einer Transaktion bzw. rückgängig machen einer Modifikation, erzeugt.
void redraw()	Methode, die bei Wechsel der Sprache aufgerufen wird und sämtliche Beschriftungen in Abhängigkeit von der aktuellen Sprache neu setzt.

void addListeners()	Methode, die die *ActionListener* den einzelnen Benutzeroberflächen-Elementen zuordnet.
void propertyChange()	Methode, die bei Änderung des *Subjects* aufgerufen wird und den *Observer* zur Aktualisierung veranlässt.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Switches zurückliefert.
void setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Switches gültigen Ergebnisdatensatz erklärt.

A.2.9 Klasse PanelCabinet

Beschreibung:

Mit der Erzeugung eines Objektes dieser Klasse werden aus dem Panel *PanelCabinet* heraus drei Sub-Panels mit den Namen *PanelCabinetImage*, *PanelCabinetList* und *PanelCabinetDetails* erzeugt. Diese wiederum repräsentieren alle für Cabinets relevanten Daten.

Attribute:

ImageIcon cabinetIcon	Bild, das die auf der jeweiligen Ebene zu verwaltenden Facilities repräsentiert.
JLabel cabinetLabel	*Label*, welches das *cabinetIcon* einbettet.
JPanel PanelCabinetImage	Sub-Panel, welches das *cabinetLabel* und *cabinetIcon* einbettet.
PanelCabinetList pcabl	Sub-Panel, welches die Liste mit Facilities vom Typ Cabinet enthält.
PanelCabinetDetails pcabd	Sub-Panel, welches die Attribute der in der Liste selektierten Facilities anzeigt.

Methoden:

PanelCabinet()	Konstruktor, der das Panel *PanelCabinet* als aktuellen Panel festlegt und die Methode *create()* für die Erzeugung der Sub-Panels aufruft.
void create()	Methode, die für die Erstellung der Sub-Panels verantwortlich ist.
PanelCabinetList getPanelCabinetList()	Methode, welche ein Objekt vom Typ *PanelCabinetList* zurückliefert.
PanelCabinetDetails getPanelCabinetDetails()	Methode, welche ein Objekt vom Typ *PanelCabinetDetails* zurückliefert.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Cabinets zurückliefert.
ResultSetListModel setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Cabinets gültigen Ergebnisdatensatz erklärt.

A.2.10 Klasse PanelCabinetList

Beschreibung:

Diese Klasse ist zuständig für die listenförmige Darstellung der im Ergebnisdatensatz vorliegenden Namen von Cabinets innerhalb eines eigenen Panels sowie für die Behandlung von Ereignissen mit der Maus.

Attribute:

Gui gui	Objekt, das die Rahmenstruktur festlegt.
ListModel listModel	Objekt, das Datensätze in Form einer Liste enthält.
ListCellRenderer renderer	Objekt, das ein selektiertes Listenelement repräsentiert.
JList list	Objekt, das eine Liste repräsentiert.
JScrollPane pane	Panel, das eine *ScrollBar* beinhaltet.
JPanel panelShelf	Panel, welches die Sub-Panels mit den Namen *PanelShelfImage*, *PanelShelfList* und *PanelShelfDetails* enthält.
String fkFieldValue	String, der den Wert des Fremdschlüssels enthält.

Methoden:

PanelCabinetList()	Konstruktor, der mittels dem übergebenen *ListModel* eine Liste mit Datensätzen von Cabinets erzeugt.
void create()	Methode zur Erstellung einer Liste mit einer *ScrollBar.*
void addListeners()	Methode, die für die Zuweisung der *ActionListener* zu den einzelnen Listenelementen zuständig ist.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Cabinets zurückliefert.

void setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Cabinets gültigen Ergebnisdatensatz erklärt.

A.2.11 Klasse PanelCabinetDetails

Beschreibung:

Diese Klasse repräsentiert die Attribute der selektierten Cabinets in der Liste. Sie fungiert als *Observer* und reagiert mit einer Aktualisierung der Attributwerte, wenn eine Änderung bezüglich des *Subjects* aufgetreten ist.

Attribute:

Gui gui	Objekt, das die Rahmenstruktur festlegt.
ResultSetListModel listModel	Objekt, das die Ergebnisdatensätze beim Auslesen von Information bezüglich Cabinets aus der Datenbank beinhaltet.
JLabel labelPosition	*Label* Current Position.
JLabel labelPositionData	*Label*, das die aktuelle Position innerhalb der baumförmigen Struktur der Datenbank anzeigt.
String positionData	Daten des *Labels*, das die aktuelle Position innerhalb der baumförmigen Struktur der Datenbank anzeigt.
JLabel labelName	*Label* Name.
JTextField textFieldName	*TextField*, das Daten bezüglich dem Namen des Cabinets enthält.
JLabel labelNumber	*Label* Number.
JTextField textFieldNumber	*TextField*, das Daten bezüglich der Nummer des Cabinets enthält.
JLabel labelSharingSwitch	*Label* Sharing Switch.
JTextField textFieldSharingSwitch	*TextField*, das Daten bezüglich dem beteiligten Switch enthält.
JLabel labelPlace	*Label* Place.

JTextField textFieldPlace	*TextField*, das Daten bezüglich dem Ort des Cabinets enthält.
JButton buttonSave	Schaltfläche, welche die Übernahme der in den *TextFields* befindlichen Daten in die Datenbank auslöst.
JButton buttonCancel	Schaltfläche, welche Modifikationen in den *TextFields* rückgängig macht.
HashMap table	Tabelle, welche für die Übernahme der in den *TextFields* befindlichen Daten in die Datenbank benötigt wird.

Methoden:

PanelCabinetDetails()	Konstruktor, der mittels der Methode *create()* für die Erzeugung der *TextFields* und Schaltflächen zuständig ist sowie diesen die *ActionListener* zuweist, um auf Ereignisse reagieren zu können.
void create()	Methode, welche die *TextFields* für die Attributwerte sowie die Schaltflächen für das Auslösen einer Transaktion bzw. rückgängig machen einer Modifikation, erzeugt.
void redraw()	Methode, die bei Wechsel der Sprache aufgerufen wird und sämtliche Beschriftungen in Abhängigkeit von der aktuellen Sprache neu setzt.
void addListeners()	Methode, die die *ActionListener* den einzelnen Benutzeroberflächen-Elementen zuordnet.
void propertyChange()	Methode, die bei Änderung des *Subjects* aufgerufen wird und den *Observer* zur Aktualisierung veranlässt.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Cabinets zurückliefert.

void setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Cabinets gültigen Ergebnisdatensatz erklärt.

A.2.12 Klasse PanelShelf

Beschreibung:

Mit der Erzeugung eines Objektes dieser Klasse werden aus dem Panel *PanelShelf* heraus drei Sub-Panels mit den Namen *PanelShelfImage*, *PanelShelfList* und *PanelShelfDetails* erzeugt. Diese wiederum repräsentieren alle für Shelves relevanten Daten.

Attribute:

ImageIcon shelfIcon	Bild, das die auf der jeweiligen Ebene zu verwaltenden Facilities repräsentiert.
JLabel shelfLabel	*Label*, welches das *shelfIcon* einbettet.
JPanel PanelShelfImage	Sub-Panel, welches das *shelfLabel* und *shelfIcon* einbettet.
PanelShelfList pshl	Sub-Panel, welches die Liste mit Facilities vom Typ Shelf enthält.
PanelShelfDetails pshd	Sub-Panel, welches die Attribute der in der Liste selektierten Facilities anzeigt.

Methoden:

PanelShelf()	Konstruktor, der das Panel *PanelShelf* als aktuellen Panel festlegt und die Methode *create()* für die Erzeugung der Sub-Panels aufruft.
void create()	Methode, die für die Erstellung der Sub-Panels verantwortlich ist.
PanelShelfList getPanelShelfList()	Methode, welche ein Objekt vom Typ *PanelShelfList* zurückliefert.

PanelShelfDetails getPanelShelfDetails()	Methode, welche ein Objekt vom Typ *PanelShelfDetails* zurückliefert.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Shelves zurückliefert.
ResultSetListModel setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Shelves gültigen Ergebnisdatensatz erklärt.

A.2.13 Klasse PanelShelfList

Beschreibung:

Diese Klasse ist zuständig für die listenförmige Darstellung der im Ergebnisdatensatz vorliegenden Namen von Shelves innerhalb eines eigenen Panels sowie für die Behandlung von Ereignissen mit der Maus.

Attribute:

Gui gui	Objekt, das die Rahmenstruktur festlegt.
ListModel listModel	Objekt, das Datensätze in Form einer Liste enthält.
ListCellRenderer renderer	Objekt, das ein selektiertes Listenelement repräsentiert.
JList list	Objekt, das eine Liste repräsentiert.
JScrollPane pane	Panel, das eine *ScrollBar* beinhaltet.
JPanel panelCard	Panel, welches die Sub-Panels mit den Namen *PanelCardImage*, *PanelCardList* und *PanelCardDetails* enthält.
String fkFieldValue	String, der den Wert des Fremdschlüssels enthält.

Methoden:

PanelShelfList()	Konstruktor, der mittels dem übergebenen *ListModel* eine Liste mit Datensätzen von Shelves erzeugt.
void create()	Methode zur Erstellung einer Liste mit einer *ScrollBar.*
void addListeners()	Methode, die für die Zuweisung der *ActionListener* zu den einzelnen Listenelementen zuständig ist.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Shelves zurückliefert.
void setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Shelves gültigen Ergebnisdatensatz erklärt.

A.2.14 Klasse PanelShelfDetails

Beschreibung:

Diese Klasse repräsentiert die Attribute der selektierten Shelves in der Liste. Sie fungiert als *Observer* und reagiert mit einer Aktualisierung der Attributwerte, wenn eine Änderung bezüglich des *Subjects* aufgetreten ist.

Attribute:

Gui gui	Objekt, das die Rahmenstruktur festlegt.
ResultSetListModel listModel	Objekt, das die Ergebnisdatensätze beim Auslesen von Information bezüglich Shelves aus der Datenbank beinhaltet.
JLabel labelPosition	*Label* Current Position.
JLabel labelPositionData	*Label*, das die aktuelle Position innerhalb der baumförmigen Struktur der Datenbank anzeigt.

String positionData	Daten des *Labels*, das die aktuelle Position innerhalb der baumförmigen Struktur der Datenbank anzeigt.
JLabel labelName	*Label* Name.
JTextField textFieldName	*TextField*, das Daten bezüglich dem Namen des Shelfs enthält.
JLabel labelNumber	*Label* Number.
JTextField textFieldNumber	*TextField*, das Daten bezüglich der Nummer des Shelfs enthält.
JButton buttonSave	Schaltfläche, welche die Übernahme der in den *TextFields* befindlichen Daten in die Datenbank auslöst.
JButton buttonCancel	Schaltfläche, welche Modifikationen in den *TextFields* rückgängig macht.
HashMap table	Tabelle, welche für die Übernahme der in den *TextFields* befindlichen Daten in die Datenbank benötigt wird.

Methoden:

PanelShelfDetails()	Konstruktor, der mittels der Methode *create()* für die Erzeugung der *TextFields* und Schaltflächen zuständig ist sowie diesen die *ActionListener* zuweist, um auf Ereignisse reagieren zu können.
void create()	Methode, welche die *TextFields* für die Attributwerte sowie Schaltflächen für das Auslösen einer Transaktion bzw. rückgängig machen einer Modifikation, erzeugt.
void redraw()	Methode, die bei Wechsel der Sprache aufgerufen wird und sämtliche Beschriftungen in Abhängigkeit von der aktuellen Sprache neu setzt.
void addListeners()	Methode, die die *ActionListener* den einzelnen Benutzeroberflächen-Elementen zuordnet.
void propertyChange()	Methode, die bei Änderung des *Subjects* aufgerufen wird und den *Observer* zur Aktualisierung veranlässt.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Shelves zurückliefert.
void setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Shelves gültigen Ergebnisdatensatz erklärt.

A.2.15 Klasse PanelCard

Beschreibung:

Mit der Erzeugung eines Objektes dieser Klasse werden aus dem Panel *PanelCard* heraus drei Sub-Panels mit den Namen *PanelCardImage*, *PanelCardList* und *PanelCardDetails* erzeugt. Diese wiederum repräsentieren alle für Cards relevanten Daten.

Attribute:

ImageIcon cardIcon	Bild, das die auf der jeweiligen Ebene zu verwaltenden Facilities repräsentiert.
JLabel cardLabel	*Label*, welches das *cardIcon* einbettet.
JPanel PanelCardImage	Sub-Panel, welches das *cardLabel* und *cardIcon* einbettet.
PanelCardList pcardl	Sub-Panel, welches die Liste mit Facilities vom Typ Card enthält.
PanelCardDetails pcardd	Sub-Panel, welches die Attribute der in der Liste selektierten Facilities anzeigt.

Methoden:

PanelCard()	Konstruktor, der das Panel *PanelCard* als aktuellen Panel festlegt und die Methode *create()* für die Erzeugung der Sub-Panels aufruft.
void create()	Methode, die für die Erstellung der Sub-Panels verantwortlich ist.
PanelCardList getPanelCardList()	Methode, welche ein Objekt vom Typ *PanelCardList* zurückliefert.
PanelCardDetails getPanelCardDetails()	Methode, welche ein Objekt vom Typ *PanelCardDetails* zurückliefert.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Cards zurückliefert.

ResultSetListModel setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Cards gültigen Ergebnisdatensatz erklärt.

A.2.16 Klasse PanelCardList

Beschreibung:

Diese Klasse ist zuständig für die listenförmige Darstellung der im Ergebnisdatensatz vorliegenden Namen von Cards innerhalb eines eigenen Panels sowie für die Behandlung von Ereignissen mit der Maus.

Attribute:

Gui gui	Objekt, das die Rahmenstruktur festlegt.
ListModel listModel	Objekt, das Datensätze in Form einer Liste enthält.
ListCellRenderer renderer	Objekt, das ein selektiertes Listenelement repräsentiert.
JList list	Objekt, das eine Liste repräsentiert.
JScrollPane pane	Panel, das eine *ScrollBar* beinhaltet.
String fkFieldValue	String, der den Wert des Fremdschlüssels enthält.

Methoden:

PanelCardList()	Konstruktor, der mittels dem übergebenen *ListModel* eine Liste mit Datensätzen von Cards erzeugt.
void create()	Methode zur Erstellung einer Liste mit einer *ScrollBar.*
void addListeners()	Methode, die für die Zuweisung der *ActionListener* zu den einzelnen Listenelementen zuständig ist.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Cards zurückliefert.

void setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Cards gültigen Ergebnisdatensatz erklärt.

A.2.17 Klasse PanelCardDetails

Beschreibung:

Diese Klasse repräsentiert die Attribute der selektierten Cards in der Liste. Sie fungiert als *Observer* und reagiert mit einer Aktualisierung der Attributwerte, wenn eine Änderung bezüglich des *Subjects* aufgetreten ist.

Attribute:

Gui gui	Objekt, das die Rahmenstruktur festlegt.
ResultSetListModel listModel	Objekt, das die Ergebnisdatensätze beim Auslesen von Information bezüglich Cards aus der Datenbank beinhaltet.
JLabel labelPosition	*Label* Current Position.
JLabel labelPositionData	*Label*, das die aktuelle Position innerhalb der baumförmigen Struktur der Datenbank anzeigt.
String positionData	Daten des *Labels*, das die aktuelle Position innerhalb der baumförmigen Struktur der Datenbank anzeigt.
JLabel labelPECCode	*Label* PEC-Code.
JTextField textFieldPECCode	*TextField*, das Daten bezüglich dem PEC-Code der Card enthält.
JLabel labelMDDate	*Label* MD-Date.
JTextField textFieldMDDate	*TextField*, das Daten bezüglich der eingestellten Herstellung der Card enthält.
JLabel labelReplacement	*Label* Replacement.
JTextField textFieldReplacement	*TextField*, das Daten bezüglich dem Austausch einer Card enthält.
JLabel labelHardwareRelease	*Label* Hardware Release.

JTextField textFieldHardwareRelease	*TextField*, das Daten bezüglich dem Hardware Release der Card enthält.
JLabel labelLiability	*Label* Liability.
JTextField textFieldLiability	*TextField*, das Daten bezüglich der Zuverlässigkeit einer Card enthält.
JLabel labelConnection	*Label* Connection.
JTextField textFieldConnection	*TextField*, das Daten bezüglich der Verbindung einer Card enthält.
JLabel labelSpareNumber	*Label* Spare Number.
JTextField textFieldSpareNumber	*TextField*, das Daten bezüglich der Anzahl vorrätiger Cards enthält.
JLabel labelSoftwareRelease	*Label* Software Release.
JTextField textFieldSoftwareRelease	*TextField*, das Daten bezüglich dem Software Release einer Card enthält.
JLabel labelPanelSide	*Label* Panel Side.
JTextField textFieldPanelSide	*TextField*, das Daten bezüglich der Panel-Seite einer Card enthält (Frontpanel, Backpanel).
JButton buttonSave	Schaltfläche, welche die Übernahme der in den *TextFields* befindlichen Daten in die Datenbank auslöst.
JButton buttonCancel	Schaltfläche, welche Modifikationen in den *TextFields* rückgängig macht.
HashMap table	Tabelle, welche für die Übernahme der in den *TextFields* befindlichen Daten in die Datenbank benötigt wird.

Methoden:

PanelCardDetails()	Konstruktor, der mittels der Methode *create()* für die Erzeugung der *TextFields* und Schaltflächen zuständig ist sowie diesen die *ActionListener* zuweist, um auf Ereignisse reagieren zu können.
void create()	Methode, welche die *TextFields* für die Attributwerte sowie Schaltflächen für das Auslösen einer Transaktion bzw. rückgängig machen einer Modifikation, erzeugt.
void redraw()	Methode, die bei Wechsel der Sprache aufgerufen wird und sämtliche Beschriftungen in Abhängigkeit von der aktuellen Sprache neu setzt.
void addListeners()	Methode, die die *ActionListener* den einzelnen Benutzeroberflächen-Elementen zuordnet.
void propertyChange()	Methode, die bei Änderung des *Subjects* aufgerufen wird und den *Observer* zur Aktualisierung veranlässt.
ResultSetListModel getResultSetListModel()	Methode, die den Ergebnisdatensatz der aus der Datenbank gelesenen Cards zurückliefert.
void setResultSetListModel()	Methode, die einen übergebenen Ergebnisdatensatz als nun für Cards gültigen Ergebnisdatensatz erklärt.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Marc Heller,
geboren am 30. November 1974 in Stuttgart-Bad Cannstatt, ehrenwörtlich,

(1) dass ich meine Diplomarbeit mit dem Titel

Entwicklung eines plattformunabhängigen Facility Management Systems unter Verwendung der JAVA Database Connectivity (JDBC)

an der **FH-Konstanz** unter Anleitung von Professor Dr. Dipl.-Ing. Reinhard Nürnberg selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als in der Abhandlung angeführten Hilfen benutzt habe;

(2) dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 18. September 2002